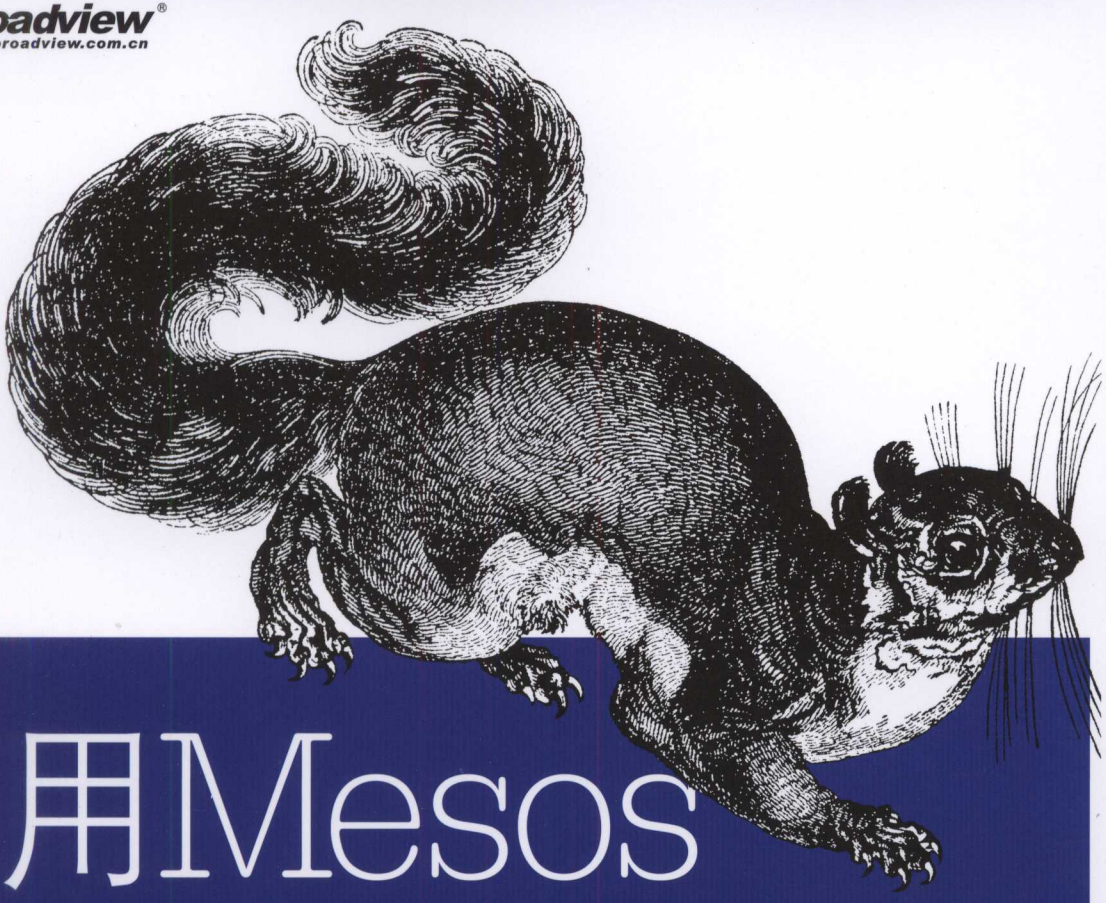
O'REILLY®

Broadview®
www.broadview.com.cn



用Mesos框架构建分布式应用

Building Applications on Mesos

利用弹性的、可扩展的分布式系统

[美] David Greenberg 著
崔婧雯 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

O'REILLY®

用Mesos框架构建分布式应用

Building Applications on Mesos

【美】David Greenberg 著

崔婧雯 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内 容 简 介

Apache Mesos是先进的集群管理器，既可以作为灵活的部署系统，也可以作为强大的执行平台。它不仅为分布式应用程序提供了良好的资源隔离，而且突破性地实现了资源的灵活共享，极大地提高了资源的整体利用率。

本书深入浅出，首先介绍了Mesos的基础知识，随后重点介绍Mesos的两种开源框架（Marathon和Chronos）。以实际程序样例为线索，一步步讲解如何配置，如何交互，以及如何构建深度集成。接着详细介绍如何为Mesos构建自定义的框架，如何构建核心Mesos API。最后深入研究Mesos的一些高级特性，比如和Docker的集成、其内部架构，以及一些最先进的API，包括数据库的持久化磁盘管理和框架预约系统。

版权贸易合同登记号 图字：01-2016-1971

图书在版编目（CIP）数据

用Mesos框架构建分布式应用 /（美）大卫 · 格林伯格（David Greenberg）著；崔婧雯译. —北京：电子工业出版社，2017.1
书名原文：Building Applications on Mesos
ISBN 978-7-121-30677-8

Ⅰ. ①用… Ⅱ. ①大… ②崔… Ⅲ. ①数据处理软件 Ⅳ. ①TP274

中国版本图书馆CIP数据核字(2016)第311330号

责任编辑：徐津平
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　邮编：100036
开　　本：787×980　1/16　印张：9.25　字数：175千字
版　　次：2017年1月第1版
印　　次：2017年1月第1次印刷
定　　价：55.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始,O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来,而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者,O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来了革命性的“动物书”;创建了第一个商业网站(GNN);组织了影响深远的开放源代码峰会,以至于开源软件运动以此命名;创立了Make杂志,从而成为DIY革命的主要先锋;公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议集聚了众多超级极客和高瞻远瞩的商业领袖,他们共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择,O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务还是面授课程,每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列(真希望当初我也想到了)非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人,他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了:‘如果你在路上遇到岔路口,走小路(岔路)。’回顾过去Tim似乎每一次都选择了小路,而且有几次都是一闪即逝的机会,尽管大路也不错。”

——Linux Journal

目录

前言

本书使用的排版约定

本书使用如下排版约定：

斜体（*Italic*）

表示新概念、URL、邮箱地址、文件名和文件扩展名。

等宽字体（`Constant width`）

用于程序列表，并且在正文内指代程序组件，比如变量或者功能名称、数据库、数据类型、环境变量、声明和关键字。

等宽斜体（*`Constant width bold`*）

表示应该由用户提供的值或者由上下文确定的值所替代的部分。

书中切口处的“▭”表示原书页码。

该图标表示技巧或建议。

该图标表示一般注释。

该图标表示警告或者注意事项。

Safari® 在线书籍

Safari 在线书籍是按需数字图书馆，以书籍和视频的格式提供来自技术和商业领域世界领先行业专家的专业内容。

技术专家、软件开发人员、web 设计人员以及商业和创作专业人士使用 Safari 在线书籍作为其搜索、解决问题、学习和认证培训的首要资源。

Safari 在线书籍为企业、政府、教育部门和个人提供一系列购买计划和定价。

会员可以在全搜索数据库里访问到上千本书籍、培训视频和正式发表前的草稿，这些资源来自 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等众多出版商。想要了解 Safari 在线书籍的更多信息，请访问网站。

如何联系我们

请将对本书的评价和发现的问题通过如下地址告知出版者。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

我们提供了本书网页，上面列着勘误表、示例和其他信息。请通过 http://bit.ly/building-applications-on-mesos 访问该页。

要给出本书意见或者询问技术问题，请发送邮件到 bookquestions@oreilly.com。

更多有关书籍、课程、会议和新闻的信息，请见网站 http://www.oreilly.com。

在 Facebook 找到我们：http://facebook.com/oreilly

在 Twitter 上关注我们：http://twitter.com/oreillymedia

在 YouTube 上观看：http://www.youtube.com/oreillymedia

鸣谢

本书的写作和出版花费了大量工作，没有大家的帮忙和支持，就没有本书最终的出版面世。

首先，感谢 Brain Foster 以及 O'Reilly 团队，他们为这本书的出版做出了很大贡献。

还要感谢我所任职的公司 Two Sigma，它给了我撰写此书的时间和支持。

感谢 Matt Adereth、Adam Bordelon、Niklas Nielsen 和 David Palaitis 的反馈和审查，这帮助大幅提高了本书的质量。

最后感谢我的妻子 Aysylu Greenberg，感谢她在撰写本书过程中给予我的爱和支持。

第 1 章

Mesos介绍

让我们坐上时光机，回溯到 1957 年。那时，使用晶体管的计算机刚刚开始出现在大学和实验室里。不过问题是，每台计算机一次只能给一个人使用。因此，大家使用纸质签到表来预约上机的时间段。因为计算机比笔和纸强大很多，因此使用计算机的需求激增。同时，计算机又非常昂贵，如果大家不能充分利用自己的预约时间，就会浪费掉上千美元！幸运的是，在那时，操作系统的概念或多或少已经开始孕育了。一个名为 John McCarthy 的天才，他发明了 LISP，想到一个伟大的主意——能否让所有用户都能将他们的工作提交给计算机，然后计算机自动在很多不同的工作间共享 CPU 资源呢？

作业成为程序

现在称为应用或者程序的东西以前称为作业（job）。在 shell 里仍然能够看到作业这个术语，如果将某个进程放到后台，就可以使用命令 jobs 来查看 shell 启动的所有程序。

一旦可以在作业间共享单台机器，就无须人工使用签到表来预约上机时间了——现在，每个人都可以很轻松地使用机器并且共享，因为机器能够按照所配置的配额，优先级，或者完全均等（如果确实需要）地执行。

飞速前进到 2010 年：随着网络化数据传输和存储费用的降低，这时已经可以将收集到的所有信息全都储存起来了。要处理所有这些数据，很可能需要使用 Storm（一种分布式实时数据处理系统）和 Hadoop。因此，你可能拥有很多机器：几台运行 Hadoop JobTrack 和 Storm Nimbus（每台都有自己独特且精细的配置），几台运行 HDFS NameNode 和 Secondary NameNode，还有 15 台机器上安装 Hadoop TaskTrackers 和 HDFS DataNodes，10 台机器用来运行 Storm Supervisor。这时就已经需要管理并且购买 30 台机器。但是，如果想将其中 5 台 Hadoop 机器改成 Storm worker，会非常困难，因

为需要将 Hadoop 机器彻底重新预配为 Storm 机器——大量实践证明，这可不像所期望的那么容易。

继续前进到 2011 年：Berkeley AMP 实验室正在起步。这正是现在给业界带来 Spark、Tachyon 和很多其他分布式可扩展系统的组织。该实验室的一个项目称为 Mesos：它意图成为能够在很多独立应用程序，比如 Hadoop 和 MPI 之间，共享计算集群的平台。在 Mesos 的论文[1]里，作者展示了所达到的卓越成果：可以在很多不同应用程序之间，包括 MPI 和 Hadoop，共享由普通计算机构成的单一集群的硬件资源。Mesos 高效地解决了集群里重新配置的问题。当然，像 Google 和 Yahoo！这样的公司对 AMP 实验室很感兴趣，因为他们也在致力于解决类似问题。Twitter 公司的团队在此领域更进一步：他们意识到 Mesos 解决了一直令他们挣扎痛苦的关键问题——如何高效使用数据中心里的大量机器——因此，他们成功雇佣了该论文的第一作者，也是 Mesos 的架构师，博士候选人 Ben Hindman，在 Twitter 公司里帮助解决这一问题。接下来的几年里，Mesos 从一个纯粹的研究项目发展成为在 Twitter 和很多其他公司里支撑成千上万台服务器的核心基础架构。

至此，大家应该对 Mesos 有所了解了（从历史角度和上下文里），一定还想知道 Mesos 到底是什么（从技术角度而言）。Mesos 是一种系统，帮助用户治理计算集群或者数据中心所有不同的机器，将每台机器当作单独的逻辑实体。使用了 Mesos 后，某个应用程序独占一组固定机器的问题就迎刃而解了。用户可以轻松调度哪个软件运行在哪些机器上，甚至可以将哪些软件运行在哪些机器上的选择权委托给 Mesos，让 Mesos 来替用户做这个决定，使得用户可以有更多时间和精力解决别的（更有意思并且更加重要的）问题。比如，在我的公司里，Mesos 帮助我们快速实践新的分布式系统，比如 Spark 和 Storm。

3 如何使用 Mesos

从最简单的角度看，Mesos 是跨集群管理 CPU、内存以及其他资源的编排平台。Mesos 使用容器化技术，比如 Docker 和 Linux Container（LXC），来达到上述目的。但是，Mesos 在此之上提供了更多的功能——它提供了实时 API，可以用来和集群交互并且辅助开发。

很多人想知道 Mesos 在其技术堆栈里所处的正确位置。Mesos 是部署基础架构的一部分吗？应该由基础架构团队管理吗？或者 Mesos 是一种应用程序开发平台，像 Heroku 那样？可能 Mesos 的确应该属于应用程序团队……不过，这些问题的答案并不直接，因

1 虽然大多数人认为第一篇论文是 Ben Hindman、Andy Konwinski、Matei Azharia 等人撰写的 NSDI 2011 论文 *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center*，但是其实 Mesos 最初发表在 2009 年的 *Nexus: A Common Substrate for Cluster Computing* 一文里，由和上文相同的作者撰写，当时称为 Nexus。

为 Mesos 所提供的功能跨越了 Infrastructure as a Service（IaaS，基础架构即服务）和 Platform as a Service（PaaS，平台即服务），以便达到系统级别上更高的效率。一方面，Mesos 是基础架构：它是用户部署 Hadoop 和 Storm 以及其他业务所需 PaaS 软件的平台。另一方面，假定用户尝试开发一种能够处理大规模计算的应用程序，类似 Hadoop 或 Spark。这时，Mesos 则又成为用户用来构建这些程序的平台。因此，与其将 Mesos 当作黑盒子，不如尝试深入研究其 API 并且学习如何为其做开发。

实际上，最好将 Mesos 当作服务于不同目标的综合体。它让用户在开发和部署应用程序时都无须再关注单台机器。它允许用户将集群看成一个单一的大型资源组。Mesos 帮助用户更为敏捷地测试并且部署新的分布式系统，当用户需要在这些流程里引入开发即运维（DevOps）团队时，能够更加快速且高效地提供部署内部应用程序的平台。对于小型项目而言，Mesos 能够容纳很多不同的系统——比如像 Ansible、Chef、Puppet 这样的部署工具现在就能够降级到基本角色，从其中引导 Mesos。Mesos 是 DevOps 的终极工具，彻底模糊了应用程序运行 API 和应用程序部署之间的界线：Twitter 只有三个运维人员，管理着成千上万个节点的 Mesos 集群。

本书探讨 Mesos 是如何完美扮演这两种不同角色的：部署系统和执行平台。

Mesos 作为部署系统

可以把 Mesos 看成一个小型部署系统。首先一起看看如今典型的部署系统，比如 Ansible 和 Chef。要使用这些工具，用户需要编写一系列任务，这些任务需要在集群里的不同机器上执行。任务可以修改文件，安装程序，并且和监管系统（比如 Supervisor 和 SystemD）交互。任务组可以组合成“角色”和“recipe”，代表更高级别的概念，比如搭建一个数据库或者 web 服务器。最终，这些任务组组成“inventory”，或者主机集，可以根据规范在其上完成配置。

这些工具比 Bash、Perl 和 SSH 有用得多，但是，它们都带有一些根本的限制。一方面，它们允许用结构化，可理解的格式编写配置信息。它们借鉴了软件开发领域的最佳实践——源码控制，封装，以及通过库函数和插件重用代码——并且将这些最佳实践扩展到系统管理领域。但是，它们最根本的设计是让一组机器遵循某个固定的配置。这样设计是刻意的：一旦用户运行了 Ansible 或者 Puppet 的配置，就希望集群处在相同的，已知的一致状态。但是，如果希望集群能够动态重新分配资源，或者根据多种外部因素，比如当前负载等，动态地更改配置，那么这些工具就爱莫能助了。

从这个角度可以认为 Mesos 是一种部署系统。分布式应用程序能够融入 Mesos 框架，从而运行在 Mesos 集群里。Mesos 不会强制用户使用某个固定的配置描述，每种框架本质

上是一种角色或者 recipe：框架包含安装、启动、监控，以及使用某种应用程序的所有逻辑。很多框架本身也是平台，当很多应用程序遵守相同的通用部署模式（比如 web 服务器）时，那么单个框架就可以管理所有这些应用程序（比如 Marathon，一种针对无状态服务的 Mesos 框架）。Mesos 的魔力在于，既然框架是一个运行着的程序，那么它就能够动态做出决策，应对工作负载以及其他集群条件的变化。比如，一个框架能够观测到有多少可用资源，然后可以改变其执行策略，以便能够基于集群条件更好地执行。因为部署系统持续运行着，它能够实时监测并且适应发生的故障，在服务器故障时自动启动新的服务器。Mesos 框架比很多传统部署系统所使用的静态描述要强大得多。

因此，可以认为 Mesos 能够代替 Ansible 或者 Chef 所提供的大多数功能。用户仍然需要一个传统的配置管理器来引导 Mesos 集群，但是，Mesos 为托管的应用程序提供了更加强大的平台：框架能够自动并且动态地适应故障机器，改变工作负载，让运维人员能够安心处理别的工作。

5 Mesos 作为执行平台

还可以把 Mesos 当作托管应用程序的平台。你可能正在使用 Heroku 运行 web 服务，或者可能管理 Hadoop 集群来执行 MapReduce 作业。在这些场景里，Heroku 和 Hadoop 集群是 web 服务和 MapReduce 作业的平台，Mesos 集群也可以作为高级别应用程序的平台。但是如果你的系统已经在工作了，为什么还要向其中加入 Mesos 呢？因为 Mesos 提供了灵活性，并且降低了被新技术赶超的风险。使用了 Mesos 之后，启动 Spark 集群或者切换到更新的技术就变得很简单，只需要启动框架，看着它引导自身就可以了。

试想一下 Heroku/PaaS 场景：Marathon 能够可靠地启动应用程序，当有机器崩溃或者关机维护时，能够自动启动新实例。Mesos-DNS 和 HAProxy 则帮助管理负载均衡（见第 3 章）。但是一旦运行了 Mesos，为什么还要操心 Hadoop 集群的维护呢？可以通过 Myriad 启动 Mesos 上的 Hadoop，甚至也不用再操心 HDFS 的管理，因为还有针对 HDFS 的框架（Mesos HDFS）。

综上，可以将 Mesos 当作一种执行平台：与其购买第三方的 PaaS 解决方案，并且为每种大数据分析技术都创建出定制集群，还不如使用单个 Mesos 集群。使用 Mesos 集群作为基础之后，用户可以在现在，或者当新需求新技术出现时，轻松启动所需的任何框架，来提供所需的任何功能。

本书是如何组织的

这里介绍本书的其他部分是如何组织的。首先，我们会一起学习 Mesos 本身，会介绍

Mesos 架构及其框架，学习如何跨集群分配资源。还会深入研究所需配置，并且介绍实用的命令行设置，来调优集群的整体性能和工作负载。

之后，我们会一起学习已有的两种开源 Mesos 框架——Marathon 和 Chronos，它们提供了应用程序托管和作业调度的功能。本书会一步步讲解如何配置它们，如何交互，以及如何构建深度集成。学习完这些框架之后，读者就能够掌握一些 Mesos 上的可靠性高且可用性强的通用应用程序架构，比如 web 服务和数据流水线。

本书后面的部分会详细介绍如何为 Mesos 构建一个自定义的框架。一起学习用 Java 实现样例作业调度框架的流程中所使用到的 API。在开发该框架的过程中，会深入探讨如何构建核心 Mesos API，有哪些常见的陷阱以及应该如何规避，如何决定是构建新框架还是采用已有系统。

本书的最后，会深入研究 Mesos 的一些高级特性，比如和 Docker 的集成，其内部架构，以及一些最先进的 API，包括数据库的持久化磁盘管理，以及框架预约系统。

本章小结

Mesos 本质上是操作系统：它管理计算机，将其统一治理成单一的逻辑单元。企业使用 Mesos 是因为它的可靠性、灵活性和高效性。Mesos 集群只要很少的管理员，就能保证大规模机器群运作良好。

框架是运行在 Mesos 上的分布式应用程序，对于它们而言，Mesos 提供了基础架构和执行服务。框架可以是应用程序，比如 Hadoop 或 Storm，也可以是部署系统，比如 Ansible 或 Chef。Mesos 和框架携手并进，将 Mesos 集群里的机器片分配给不同的角色。这样的能力——将某个框架所独占的机器群壁垒打破——正是 Mesos 带来更高运维效率的秘密所在。

本章介绍了 Mesos 的高层级视野，下一章我们开始深入 Mesos 概念，并且在实践中开始 Mesos 的学习！

第2章

开启Mesos之旅

现在大家已经从高层理解了 Mesos 所能解决的问题类型，本章会继续深入探讨。项目中使用什么术语？ Mesos 如何管理资源？ Mesos 给应用程序强加了什么结构化的约束？通过回答这些以及其他问题，我们就能够逐渐理解 Mesos。接下来的小节里，本书会具体介绍其中的所有技术，帮助大家学习使用 Mesos 构建应用程序所需了解的技术。

框架

在 Mesos 的世界里，假定所用的分布式系统拥有中央控制器和很多其他的工作单元。这是大多数成功的可扩展系统的工作方式：Google 的 BigTable、Apache Hadoop，以及 Twitter 的 Storm 都拥有这样的中央控制器来组织管理工作单元。当然，工作单元必须设计成可以独立于控制器进行工作，这样它们就不会被控制器的瓶颈所影响，同时不依赖于控制器就能够实现高可用性。运行在 Mesos 上的应用程序称为框架。一个框架由两部分组成：控制器部分，称为调度器；工作单元部分，称为执行器。

在 Mesos 集群上运行框架，必须运行其调度器。调度器就是能够以 Mesos 协议通信的一个简单进程。通常，调度器在 Marathon 上运行，但是有必要考虑如何确保调度器的高可用性。当一个调度器首次启动时，会连接到 Mesos 集群上，这样它才能使用集群的资源。调度器运行之后，会给 Mesos 发送请求，启动它认为合适的执行器。调度器的百分之百灵活性正是Mesos如此强大的原因所在：调度器能够基于资源的可用性、改变的工作负载，或者外部触发器来启动任务。现在，业界有能够管理海量 web 服务的调度器，能够协调 Storm 拓扑的调度器，以及能够优化批量作业替换的调度器。

当某个调度器想完成某些工作时，就会启动执行器。执行器仅仅是调度器的工作单元：调度器随后决定给执行器发送一个或多个任务，执行器会独立完成这些任务，任务完成后，向调度器发送状态更新。本书在“任务和执行器”部分会详细介绍任务和执行器间

的区别。

至此我们已经了解了框架是什么，接下来一起学习 Mesos 集群本身。

Master 和 Slave

Mesos 集群由两个组件组成：Mesos master 和 Mesos slave。master 是协调集群的软件，slave 则在容器里执行代码。你可能会想，“这听上去和框架的工作机制很类似”。

在一些情况下，这么想 100% 是正确的。Mesos 集群和其上运行的框架的结构一样，因为 master/slave 模式很简单，灵活并且高效。但是，集群软件需要隔离开，来确保框架之间保持独立和安全。这里先介绍 Mesos master 的一些核心概念，随后介绍 slave。

Master

Mesos master 是集群的大脑，它们的职责如下：

- 它们是运行任务的中央资源。
- 它们在所有连接上的框架间公平共享集群。
- 它们为集群维护主要的 UI。
- 它们确保资源的高可用性和高效分配。综合考虑，它们会成为单点故障点，但是 Mesos 会自动处理这种情况，能够从单个 master 的故障里恢复。

让我们一起将这些职责逐一分解，看看对于 master 的设计而言，它们意味着什么，又是如何影响集群的：

Master 是运行任务的中央资源

为了让 master 能够以最小延迟提供 UI 以及任务相关的数据，所有任务元数据都驻留在内存里。这意味着运行 master 的节点必须拥有足够的内存——比如，在拥有上千并行任务的集群上，master 可能需要消耗 20GB RAM。注意这并不包括已完成的任务，对于一些工作负载而言，会释放已完成任务的内存，从而减轻 master 的压力。master 负责存储和最近完成的任务相关的元数据，但是为了节约内存，它们拥有固定大小的缓存来存储最近的成百上千条任务。在一个高吞吐量的集群里，这意味着完成任务的元数据在 master 上可能仅仅能用几分钟。要重新获得更长时间之前所完成任务的元数据，可以使用 Mesos 的监控系统 Satellite，自动将任务元数据复制到 NoSQL 数据库里。

Master 在所有连接上的框架间公平共享集群

master 负责给框架提供资源。因为 master 准确地知道哪个框架使用哪些资源，所以它们能够知道哪种框架使用的资源比配额少，并且让这些框架能够优先占用空闲资源。[1] 当配置 Mesos 集群时，有很多种选择来控制所连接上的框架之间分配资源的方式。框架连接 Mesos 时，都是通过角色连接。每种角色能够拥有为之特别定制的资源，并且可以给每种角色分配集群剩余资源的不同权重。集群管理员可以配置角色，确保满足每种框架的 SLA（详见“角色”部分）。

Master 为集群维护主要的 UI

使用浏览器检查并且和 Mesos 集群交互时，通常会首先访问 master UI，默认使用端口 5050。该 UI 让用户可以看到集群里总体还有多少可用资源，以及已经使用了多少。还可以看到所有连接上的框架及其信息，包括框架 ID，以及注册到 Mesos 上的时间。还可以看到所有当前资源 offer 和任务的细节信息。master UI 自动将每个任务和运行该任务的 slave 的 UI 链接到一起，因此还可以从任务的沙箱里访问文件，比如其日志。

Master 确保资源的高可用和高效分配

Mesos 意在简化高可用性集群的构建和运行。这也正是需要运行三个或者五个 master 的原因：只要大部分 master 还在运行，集群就可以继续照常运行。master 使用业界公认的 Paxos 算法，来持续同步集群状态。因为所有 master 都有相同的最新视图，所以任意 master 都能够服务并且处理请求。同时，Mesos 试图构建高度可扩展的集群，也就是说要能够快速管理上千个集群的分配。要达到这一目的，一次只会选举出一个主实例。这个主实例负责高效运算，并且作出资源分配决策，从而保证决策不会因为等待其他 master 的回应而变得迟缓。这样的设计使得 Mesos 能够达到高速率的可扩展资源分配，同时还能够在有部分故障时提供高可用性。当然，大部分 master 发生故障时会强制 Mesos 集群进入“安全模式”——大部分 master 下线后，框架就无法分配新资源或者启动新任务。但是，任务的运行仍然能够继续。这意味着即使 master 的运行中断，只要所使用的框架仍然能够继续工作，无须启动新的任务，那么整个集群就仍然能够在单组件故障时提供不中断服务。

本书不会讲解如何安装以及运行 Mesos master 和 slave：网上有大量教程详细介绍了这些，并且可以根据用户偏好的操作系统和部署系统做定制。

1 Mesos 使用一种公平共享分配算法，称为 Dominant Resource Fairness（DRF），参阅 Ali Ghodsi 等在 2011 年撰写的论文“Dominant Resource Fairness: Fair Allocation of Multiple Resource Types”。

Slave

Mesos 1.0 里，slave 会更名为 agent

Mesos 1.0 发布时，会改动一些术语——Mesos slave 更名为 Mesos agent，并且所有出现“slave”的地方都会变更为“agent”。

Mesos master 的补充就是 slave。slave 的职责有所不同，如下所示：

1. 它们启动并且管理托管执行器的容器（LXC 容器和 Docker 镜像）。
2. 它们提供 UI 来访问容器内数据。
3. 它们和本地执行器通信，从而管理和 Mesos master 的通信。
4. 它们展示所运行的主机的信息，包括运行任务和执行者的信息、可用资源，以及其他元数据。
5. 它们管理任务的状态更新。
6. 它们创建状态的检查点，从而启用集群的滚动重启。

下面逐一详细介绍。

容器管理

Mesos 为大型集群提供最为先进的多租户隔离功能。该功能借助于容器化技术，容器是一种轻量级的，资源消耗极低的方式，用来确保特定程序能够访问特定数量的资源。比如，无论同一台机器上的其他应用程序有什么要求，容器都可以保证其内运行的 web 服务器能够一直使用 2 个 CPU，1GB 的 RAM 和 500 MBps 的网络带宽。Mesos slave 和 LXC 以及 Docker 深度集成，LXC 是标准的 Linux Containers 技术，Docker 则是最流行的打包和容器化技术。因为 slave 需要能够启动容器，并且确保能够以特定用户运行执行器，所以通常使用 root[1] 运行 slave 进程，这样它就能够创建容器，并且以不同的用户运行不同的执行器。

11

查看 slave 特定数据的 UI

正如 Mesos master UI 展示整个集群状态的信息，比如所有活跃任务和框架，slave

1 可以用某个特定用户运行 slave，那么 Mesos slave 仅仅可以用该用户运行执行器。这在高安全性要求环境里，或者开发过程中非常有用。

UI 则展示某个特定 slave 所发生事情的信息。除了展示 master UI（仅仅显示框架、任务和与 slave 相关的执行器）的更为详细的版本，slave UI 还允许用户浏览容器内的所有文件。这一功能让用户可以很容易地查看任何正在运行或者已经完成任务的日志，这对于调试以及监控集群而言至关重要。通常 Mesos 框架 UI 会链接到 slave 的 UI 上，从而让用户能够轻松浏览、查看并且下载任务所生成的文件。

和执行器以及 master 通信

你很可能正致力于（或者将要从事）分布式系统的工作。如果是这样的话，你应该已经知道（或者很快就会发现）分布式系统工作很难做。为了帮助普通程序员解决这些难题，Mesos slave 充当了执行器和 master 之间的通信员。这意味着执行器和 slave 之间的所有通信都是在本地完成的，基于 100% 可靠的本地回路接口，从而用户无须担心分布式系统的可靠性。[1] 因此，Mesos slave 负责创建检查点，并且在网络有问题或者重启时负责状态的重新同步，这有助于降低构建可靠 worker 应用程序的难度。

告知宿主机器的信息

在 Mesos 里，每个 slave 负责将其自身能提供来运行任务的资源数量汇报给 master。这样的设计使得必须给 Mesos 集群添加如下新功能：一旦启动了某台 slave 机器，并且 Mesos slave 进程连接到 master 上，slave 就得选择汇报集群可用哪种资源。默认情况下，slave 进程会自动监测机器上的 CPU、内存和硬盘，并且将这些都暴露给 Mesos，所有角色都可以使用。（slave 会预留 1GB 和所检测到内存的 50% 中值比较小的内存，来运行自身以及其他操作系统服务。同样，还会预留 5GB 和检测到的磁盘的 50% 中值比较小的磁盘容量。）但是，用户可以在命令行指定必须暴露多少资源，以及是否必须为一个或者多个角色预留某个或者所有这些资源。除了自动检测到的资源，管理员还可以配置 slave 暴露自定义资源，这在“配置自定义资源”部分会详细介绍。一个 slave 还可以暴露有关机器信息的任意键 / 值数据（详见“配置 slave 属性”部分）。这是一种很好的方式，可以向一些调度器传递有用信息，比如机器在哪个机架上，支持哪一代处理器特性，或者运行的是什么操作系统。通过加强 slave 和集群共享的这些信息，用户可以在自己的调度器里添加强大的特性。比如，在“使用位置约束变量”部分，可以看到如何使用这些信息将 Marathon 的任务分发到位于待定机架的机器上。

12

状态更新

当某个任务向其调度器发送状态更新时，需要保证至少交付了一次。slave 负责在

1 这并不一定正确。记住，分布式系统很难！

本地磁盘上存储所有的状态更新，并且当任务调度器没有得到这些信息时，会周期性重新发送信息。不这样设计的话，在发生短暂的网络或调度器故障时，框架可能就无法知道其任务是何时完成的。

滚动重启

构建永远在线的分布式系统时会遇到一些需要技巧才能解决的问题：在不导致下线的情况下能够完成升级。通常来说，用户可能会在某个时间点尝试重启 slave 集群内的一小部分机器——这样，可以始终保持 90% 或 95% 的集群处理能力。不幸的是，这样的策略有一个问题。想想那些运行在内存缓存里的 slave，即使一次只重启一台机器，也会需要很长时间才能重新注入这些缓存，从而导致必须在如下选择中二选一：更快地完成 slave 的升级，同时忍受比升级本身时间长得多的应用程序性能下降，或者慢慢升级 slave，并且增加通过大小或者其他顺序滚动升级所需的时间。Mesos slave 提供了称为创建检查点（checkpointing）的特性，这让用户有了第三种选择：slave 能够周期性地将自身状态及其任务和执行器的状态写入快照，这样当重启 slave 进程时，就不需要“杀死”所有任务和执行器，slave 新的升级后版本可以简单读取检查点文件，通过重新连接到已有执行器上，从而接替之前版本的工作。这样，就可以快速完成 slave 的滚动重启，并且不需要终止任何单个任务。

在 Mesos slave 上使用 Cgroup

Cgroup，是控制组（control group）的简称，它将 Linux 进程组织在一起从而完成资源审计和分配。每个 Mesos slave 都必须启用 cgroup，因为 cgroup 所提供的 CPU 和内存隔离最终保证了 Mesos 的可靠性。Linux 系统里的进程属于 cgroup 层级里的节点；层级里的每个节点拥有一部分资源，节点的每个子节点拥有该节点资源的一部分。通过这样的资源层级，从 CPU 份额、内存到网络和磁盘 I/O 带宽都能够被计算并且精确共享。这样的共享和计量保证某个任务不会耗尽另外任务的资源。当 Mesos slave 使用 cgroup 时，实际上会在层级里动态创建并且销毁节点来匹配 master 计算出的份额。这样做简化了管理员的工作，因为他们仅仅需要确保 slave 的 OS 里启用了 cgroup 功能即可。

在 Debian wiki 和 Red Hat Resource Management Guide 里可了解如何启用 cgroup。Mesos slave 在 Debian 和 Red Hat 7 的默认 cgroup 配置下就能够自动工作。

要在 Mesos slave 上启用 cgroup 隔离，必须向每个 slave 传入命令行参数 `--isolation='cgroups/cpu,cgroups/mem'`。在生产环境里不推荐使用默认的

> Posix 隔离器，因为它无法提供和 cgroups 隔离器一样健壮的隔离保障。Posix 隔离器的主要用途是当开发人员在 Mac 上为了开发代码而运行 Mesos 时，能够得到进程监控数据。

现在我们已经理解了 Mesos slave 的关键职责。slave 的根本目标是暴露集群资源，下一节会介绍这些资源是什么，以及如何定义自定义资源。

资源

Mesos的最根本抽象是资源——任务和执行器在完成工作时消费资源。大家通常会问："什么是资源？"答案很宽泛：资源是任务或者执行器工作期间使用的任何东西。比如，标准资源（也就是几乎每种框架都需要使用的资源）是 `cpu`、`mem`（内存）、`disk` 和 `port`。前三种资源是标量：它们表示可用的总量，从其中减去任务的使用量。当然，Mesos 里还有其他类型的可用资源。范围（Range）是一个整数列表，其中单个元素或者某个子集能够被分配。比如，范围可以用来表示可用以及正在使用中的端口。最后一种资源类型——集合（set）——是一些字符串的乱序集合。后面会介绍一个集合的样例，这里先介绍标准资源：

`cpu`

这一资源表示有多少可用的 CPU 核。任务可能会占用 CPU 的一部分——这是很有可能的，因为 Mesos slave 使用 CPU 共享，而不是独占指定的 CPU。这意味着，如果预留了 1.5 个 `cpu`，那么进程每秒总计可以使用 1.5 秒的 CPU 时间。这也意味着，在单个执行器里，两个进程，每秒可以得到 750 微秒的 CPU 时间，或者在 1 秒内，一个进程得到 1 秒 CPU 时间，而另一个得到 500 微秒 CPU 时间。使用 CPU 共享的优势是如果某个任务能够利用比自己的配额更多的资源，并且没有其他任务使用空闲的 CPU，那么这个任务很可能就能使用比配额更多的资源。这样，`cpu` 预留保证了任务可用的最小CPU时间——如果有可用的额外资源，也允许它使用更多资源。

`mem`

`mem` 表示可用内存有多少兆。和 CPU 不一样，如果有空闲 CPU 资源就可以使用比配额更多的 CPU 时间，内存则是严格预分配的资源。因此，为任务选择正确的内存很有技巧：太小的话，当任务尝试分配比配额多的内存时，就会被"杀死"；太大的话，则会导致太多的空闲内存，而其原本可以用来运行更多的任务。

disk

disk表示容器沙箱里可用的空间总量（以兆计）。因为磁盘的使用可能大幅度受不可预见的错误日志的影响，很多框架开发人员并不在其任务里预留磁盘资源。而且，Mesos slave在任务使用超过分配的磁盘时，默认并不会强制执行磁盘的限制条件（要改变该默认行为，可以在slave的命令行里传入参数--enforce_container_disk_quota）。忽略该值的另一个原因是Mesos slave会在需要回收再利用磁盘空间时自动回收已完成任务的沙箱（下面侧栏提供了为slave配置垃圾回收的细节和建议）。

15

配置slave沙箱垃圾回收

当任务在Mesos集群上运行时，它们肯定都会向磁盘写入数据。任务会生成日志、临时存储输出文件，或者缓存数据及二进制文件。如上所述，大多数Mesos用户都无须费心启用磁盘配额限制，因为大部分磁盘都很大，单个作业不太可能将磁盘写满。但是，这样的逻辑只在短期内起作用。任务运行数天或者数周后，在某个机器上的输出之和肯定会超出slave的磁盘能力。要解决这个问题，Mesos slave会从磁盘垃圾里回收旧数据。如下所示是配置slave磁盘垃圾回收器（GC）的命令行参数：

--gc_delay

该参数指定执行器目录在其被删除前能够存在的最长时间。比如，可以设置为--gc_delay=3days或者--gc_delay=2weeks，默认值--gc_delay=1weeks。注意系统可能会缩短该延迟时间，这取决于可用磁盘的使用情况。

--disk_watch_interval

该参数确定多久检查一次磁盘使用（以及GC多久运行一次）。它使用和GC延迟设置一样的时间符号。比如，可以设置为--disk_watch_interval=10secs；默认值是--disk_watch_interval=1mins。如果设置得太过频繁，就可能会太过频繁地枚举目录而影响到硬盘性能。

--gc_disk_headroom

该参数决定GC能够回收多大比例的headroom，或者空闲空间。如果将headroom设置过小，某个任务过快地写入磁盘，可能会在GC运行前耗费掉所有可用的空间。另一方面，如果设置过大，则会在垃圾必须回收之前就过早地回收了文件，而妨碍到调试（比如，在查看前数据就已经被删除了）。该参数

参数的值必须在 0.0 到 1.0 之间；默认值是 `--gc_disk_headroom=0.1`，也就是说 Mesos 认为 10% 的磁盘应该是空闲空间。用来计算某个输出目录的可存活时间的公式是：

```
max_age=max (0.0, (1.0 - gc_disk_headroom - disk_usage))
```

`port`

port 是用户需要理解的 Mesos slave 上第一个（也是唯一一个默认的）非标量资源。很多任务，比如 HTTP 服务器，会尝试开启一个或者多个端口，从而可以监听入站链接请求。因为每个端口只能绑定到一个任务上，Mesos 允许任务预先占用特定端口，这样每个任务使用的端口范围不会互相冲突。按默认来说，Mesos slave 暴露端口范围为 31000 ~ 32000，但是，可以在命令行里通过设置 `port` 资源来改变这个范围值。

至此，本书已经介绍了 Mesos slave 资源的基本概念及其默认行为，下面介绍如何自定义这些资源。

配置自定义资源

用户可能想要增加或者减少操作系统服务所使用的资源集，可能需要暴露额外端口，或者特定范围内的端口，来运行某个框架。slave 机器可能带有外部类型的硬件，比如 GPGPU，从而需要 Mesos 管理这些设备的预留和分配。最终，用户可能想要将某些资源预留给某个特定角色使用。在这些情况下，必须向 Mesos slave 提供自定义的资源配置。

Mesos 资源由键 / 值对指定。键是资源名称，比如 `cpu`、`mem`、`disk` 或者 `port`。值是这些资源有多少可用。资源类型由值的语义来决定。每种资源记录成键：值，所有资源由分号串联起来，通过命令行传入。Mesos 支持标量、集合和范围资源。在 Mesos 文档里可以查看资源属性的完整语法，本书在下例里简单介绍。

假定需要 slave 通报如下资源：4.5 CPU，8GB RAM，端口范围在 1000 ~ 5000 和 31000 ~ 32000。并且拥有 2 个 GPGPU。那么，需要设置如下：

资源	值	注释
`cpu`	`4.5`	注意标量资源可以提供 double 值
`mem`	`8192`	内存以 MB 为单位
`port`	`[1000-5000,31000-32000]`	范围记录在方括号内；用逗号连接不同的范围
`gpgpu`	`{gpu1,gpu2}`	记录在花括号内；元素由逗号隔开

在命令行传入，使用如下参数：

```
--resources='cpus:4;mem:8192;ports:[1000-5000;31000-32000];gpgpus:{gpu1,gpu2}'
```

17 所有这些指定的资源都属于默认角色。

配置 slave 属性

slave 可以将机器的任意键 / 值数据暴露成属性。和资源不一样，属性不由 Mesos 分配器使用。它们仅仅传递给框架调度器来辅助其调度决策。假定需要将下述键 / 值对表格传递为 slave 的属性：

键	值
operating_system	Ubuntu
cpu_class	Haswell
zone	Us_east
rack	22

要将这些属性传递进 slave，需要使用冒号分隔开键和值，用分号分隔开每个键值对。最终传入的命令行参数如下：

```
--attributes='operating_system:ubuntu;cpu_class:haswell;zone:us_east;rack:22'
```

角色

为了决定哪些资源能够提供给哪些框架，Mesos 使用了“角色”的概念。角色类似于组：用户可以有 dgrnbrg 角色（给运行的东西所用），qa 角色（给质量保证相关的任务所用），db 角色（给数据库所用），等等。除了可以显式预留资源，角色还能够用来给 Mesos 的公平分享算法指定权重。用户可以指定一些角色比其他角色能够接收到集群里更大比例的资源。当注册框架时，可以通过设置注册所用的 FrameworkInfo 消息里的 role 字段，来指定其属于哪种角色。框架只能够接收到属于其角色，或者 * 角色的 offer。

如果用户想要使用角色，那么必须在 master 上设置由逗号分隔的有效角色列表。比如，如果想要设置 dev、qa 和 prod 角色，需要向 master 传递 --roles=dev,qa,prod 参数。给这些角色分配权重，让 dev 可以得到 qa 两倍的 offer，prod 能够得到 qa 三倍的 offer，需要传递 --weights='dev=2,qa=1,prod=3' 参数。

保护角色

任何框架默认可以声明属于任意角色。如果需要，可以通过访问控制列表（ACL）来保护角色，ACL是一种安全特性，不在本书探讨范围之内。`register_frameworks` ACL控制在注册框架时，每个principal（也就是用户）能够使用哪些角色。

静态和动态slave预留

slave预留用来确保特定资源始终能够用于特定的用户场景。直到Mesos 0.25版本，运维人员都只能使用静态预留方式。现在，也提供了动态API来预留以及解除预留资源，而不再需要改变命令行。

静态预留

除了在命令行里指定某个slave暴露哪种资源，用户还可以指定哪些角色可以使用这些资源；这样可以为某个特定角色预留所需资源。本书上个示例里没有为资源指定任何角色，因此，所有资源都被分配给默认角色。默认角色是通过slave命令行参数`--default_role`来指定的。如果没有设置`--default_role`，那么默认角色就是*（任何人）。

为某种特定资源指定角色时，需要将角色名字放在资源后的圆括号里。假定机器有4个CPU以及16GB内存。用户想要为角色prod预留1个CPU和4GB内存，为角色`qa`预留2个CPU和1GB内存。系统的任何用户都可以使用剩余的资源。那么，相应设置如下：

资源	值	角色
cpus	1	prod
mem	4096	prod
cpus	2	qa
mem	1024	qa
cpus	13	*
mem	11264	*

配置slave所使用的命令行参数如下（注意这些参数需要在单行内键入，不要添加\或者任何空格；这里命令折行是因为图书宽度所限）：

```
--resources='cpus(prod):1;mem(prod):4096; \
  cpus(qa):2;mem(qa):1024;cpus:13;mem:11264'
```

注意无须特别为大家都能使用的资源指定角色*，因为这是默认角色。还可以将默认角

色改动为 prod，命令行则应该修改如下：

```
--default_role=prod \
  --resources='cpus:1;mem:4096;cpus(qa):2;mem(qa):1024;cpus(*):13;mem(*):11264'
```

注意更改默认角色意味着自动检测到的磁盘和端口范围都会赋给 prod 角色。

如上所述，slave 预留是十分强大的系统，让用户可以进行控制，从而确保生产环境上至关重要的工作负载总是能够获得满足其需要的资源分配。同时，它们的配置相对枯燥，修改相对困难，因为需要改动 slave 的命令行。

动态预留

从 Mesos 0.25 开始，可以通过 `/master/reserve` 和 `/master/unreserve` HTTP 端点来创建以及销毁预留。本节探讨如何预留以及解除预留上节示例里的资源。假定已经为用户管理员 `admin` 配置了 HTTP 基础授权，并且配置了合适的 ACL。下面首先编写一些 JSON，指定需要预留的资源（参见示例 2-1）。

示例 2-1　resources.json

```
[
    {
        "name": "cpus",
        "type": "SCALAR", ❶
        "scalar": { "value": 1 },
        "role": "prod",
        "reservation": {
            "principal": "admin" ❷
        }
    },
    {
        "name": "mem",
        "type": "SCALAR",
        "scalar": { "value": 4096 },
        "role": "prod",
        "reservation": {
            "principal": "admin"
        }
    },
    {
        "name": "cpus",
        "type": "SCALAR",
        "scalar": { "value": 2 },
        "role": "qa",
        "reservation": {
            "principal": "admin"
        }
    },
```

```
    {
        "name": "mem",
        "type": "SCALAR",
        "scalar": { "value": 1024 },
        "role": "qa",
        "reservation": {
            "principal": "admin"
        }
    }
```

❶ HTTP API指定资源的方式和指定Mesos资源protobuf的方式相同。参考“理解mesos.proto”部分了解该结构。

❷ 动态预留必须和某个用户或者管理员关联。要完成预留，需要在提交时出示admin用户的证书。

要提交预留，还需提供想要预留该资源的slave的slave ID（本示例中使用$SLAVEID，一般slave ID类似于201509191529-2380865683-5050-11189-4621）。下例展示如何将用户证书，resources.json和slave ID整合到curl命令里：

```
curl -u admin:password \ ❶
     -d slaveId=$SLAVEID\ ❷
     -d resources="$(cat resources.json)"\ ❸
     -X POST\ ❹
     http://<ip>:<port>/master/reserve ❺
```

❶ 本例假定admin用户的密码为password。

❷ 以键/值对的格式将slave ID和资源提供给端点。

❸ 以JSON的格式将资源传递给curl。注意Mesos实际得到的数据类似于slaveId=$SLAVEID&resources=$JSON。

❹ /reserve和/unreserve端口都要求POST数据。

❺ 需要指定Mesos master的URL。使用该API可以更加方便地配置预留，因为用户仅仅需要和master交互，而不需要直接和slave交互来配置每个slave的预留。 21

销毁预留几乎和创建预留一样：

```
curl -u admin:password \
     -d slaveId=$SLAVEID\
     -d resources="$(cat resources.json)"\
     -X POST\
     http://<ip>:<port>/master/unreserve ❶
```

❶ 注意唯一不同之处在于，现在使用的是 /unreserve 端口，仍然需要精确指定需要解除哪些资源的预留。

使用这个新的 slave 预留的 HTTP API 能够更为容易地创建、管理及销毁实时预留。随着使用量的增加，和该 API 交互的工具迅速增多（请记住该功能于 2015 年 10 月份发布）。

任务和执行器

调度器想要完成某些工作时，会启动一个执行器。执行器可以简单理解为调度器的 worker：调度器决定给某个执行器发送一个或者多个任务，执行器会独立完成这些任务，在任务完成时将状态更新发送给调度器。

到这里，本书还没有详细介绍任务和执行器这两个概念的区别，因为通常每个执行器只有一个任务，因此没有太大必要严格区分这两者。但是，一些框架高度利用了任务和执行器的抽象，要想理解这些框架，就必须首先理解这些抽象的概念。首先需要理解这个问题：任务是调度器想要在某个 slave 的容器里运行的东西。但是，有时候用户想要在同一个容器里运行多个任务。比如，Storm worker 和 Hadoop TaskTracker 都体现了这样的使用模式，这两者都可以包含很多任务。

那么，什么是执行器？执行器是运行任务的进程容器。

什么是任务？任务是 Mesos 里的工作单元。

图 2-1 展示了该层抽象里的组件之间的关系。

22

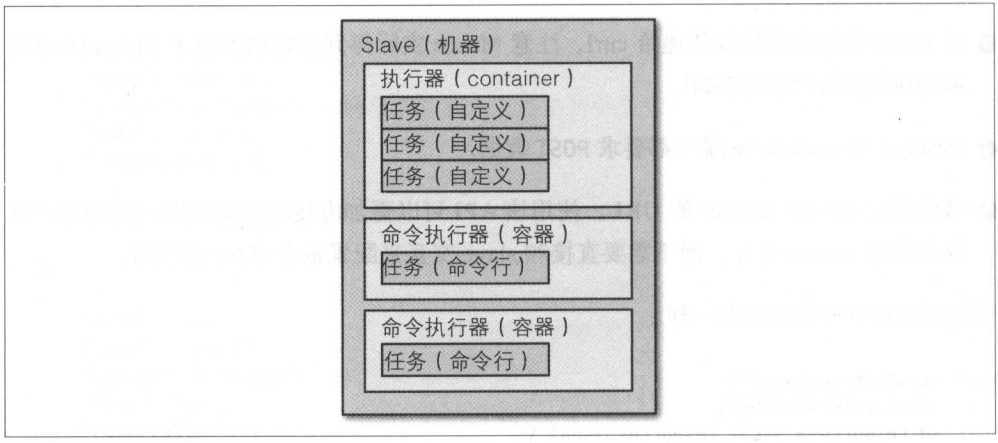


图2-1 slave（机器）、执行器（容器）和任务之间的关系

还需要理解如下概念：

任务能够调整执行器容器的大小

> 当用户在 Mesos 里启动一个任务或者执行器时，这两者都带有附加其上的资源。这样设计的原因是执行器带有自身运行所需的基本资源，并且每个任务都可以使用额外资源。Mesos 会自动扩展和收缩执行器的容器，使其拥有的资源等于执行器以及所有其运行任务所需的资源总和。

API 可以面向任务或者执行器

> 当为执行器 API 编程时（细节参见第 5 章），一些执行器和调度器之间的通信 API 是基于任务（比如 `launchTask` 和 `statusUpdate`）指定的，而另外一些 API 是基于执行器（比如 `frameworkMessage`）的。框架的开发人员确保在需要时将消息和任务关联起来，而不管 Mesos API 是否已经自动完成了。

对大部分框架而言，无须区分任务和执行器

> 大多数 Mesos 框架都是一个执行器一个任务。也就是说用户无须担忧执行器 API 的多样风格。不幸的是，这反过来也意味着执行器 API 有时候很难理解，因为它们用得很少。

CommandExecutor

大多数用户和开发人员不需要使用 Mesos 执行器 API 的复杂特性。为了快速并且可靠地开发应用程序，Mesos 提供了 `CommandExecutor`，这是一种特别的简化执行器。如果任务仅仅简单地启动一个命令行应用程序（也可能是从存储的某个位置下载的）或者一个 Docker 的镜像，就可以在启动任务时指定 `CommandInfor`，而不是 `ExecutorInfo`。如果这么做，那么 Mesos 会使用内建的 `CommandExecutor` 启动特定命令，会为用户处理 Mesos 的消息协议。很多框架，比如 Chronos 和 Marathon，为了简洁性通常使用 `CommandExecutor`。

理解 mesos.proto

在阅读本书时，以及你在自己尝试开发时，会经常想知道如何配置一些 API，或者想要知道某些特性的配置选项有哪些。所有和 Mesos 的通信使用的都是 Google Protocol Buffer（也称为 protobuf）编码。幸运的是，所有数据模式都存储在同一个地方：Mesos 代码基的 include/mesos/mesos.proto 文件里。本节介绍 Google 的 protobuf，并且学习 Mesos 代码基里会频繁出现的一些术语。

protobuf是一种接口描述语言，让应用程序开发人员能够指定其系统里存在的强类型消息。很多语言都带有protobuf编译器，可以将.proto文件转换成高效序列化器和反序列化器，因此所有语言都可以和高效的二进制编码通信。

首先介绍protobuf的基本概念：

```
/** ❶
 *  这里是用户信息。
 *  本书尽可能尊重隐私。
 */
message UserInfo { ❷
  // 用户名
  required string username = 1; ❹
  required Name name = 2; ❺
  optional uint32 age = 4; ❻
  // 辅助信息
  optional bytes aux_info = 3; ❼
}

message Name { ❽
  optional string first = 1; ❾
  optional string last = 2;

  message MiddleName { ❿
    optional string regular = 1; ⓫
    optional string patronymic = 2;
  }

  repeated MiddleName middle = 3; ⓬

  optional uint32 total_parts = 4 [default = 2]; ⓭
}
```

❶ protobuf支持JavaDocs风格的块注释。

❷ proto文件的主要语法格式是message，名称（UserInfo）以及一系列指定类型和名称的字段。

❸ protobuf还支持C和Java那样的可以扩展到行末的注释。

❹ 字段包含四部分：是requried、--optional还是repeated；类型（本例中是string）；字段名称；字段索引。每个字段索引在消息里必须是唯一的，开发人员需要负责确保永远不重用索引号，因为这样能够确保构建出的二进制格式具有后向兼容性。

❺ 字段类型可以本身是另一个message。

❻ protobuf有很多内建类型，Mesos主要使用int32、uint32和double。也要注意字段索引无须遵守数字顺序——通常将它们组织起来反映消息的结构。

❼ bytes允许用户在protobuf里存储任意二进制数据。在Mesos的protobuf消息里，这是用来让开发人员定义自定义的额外工作负载的。

❽ 通常来说，大部分或者所有项目消息都定义在单个文件里。

❾ 不需要特别将任何字段标记为required。

❿ 消息可以嵌套进另一个消息。

⓫ 注意，每个消息，即使嵌套的消息，也会重新开始字段计数。

⓬ 字段也可以重复，可以是一列零，或者特定类型的入口（本例中是MiddleName）。

⓭ protobuf允许指定默认值。

除了这些基础知识之外，mesos.proto里还会经常出现一些模式。首先，系统里的任意ID都有一个为之定义的消息，只有一个required的string值（比如FrameworkID、TaskID）。这对于类型化语言很有利，比如C++、Scala和Java，因为这样确保了不同类型的ID不会混成一团。第二，当可以选择指定什么时（比如，自定义执行器vs.命令行执行器），这些选择都是optional的，并且在mesos.proto里必须加上注释解释这些选择。框架代码能够生成的大部分消息都有一个类型为bytes的data字段：这里可以放置想要传输的任意序列化消息。

25

我从来不关闭的浏览器标签页

基于对该模式注释的信赖，我在开发框架代码时会一直打开mesos.proto。

mesos.proto里的另一个常见模式是标记union。标记union也称为辨别union；如果配置的某个东西有多个互斥选项时就需要使用它。这一类型的union，比如Resource消息，每个选项都是optional的，这样用户可以每个消息填入一个。还有个required的Type字段，这向代码表明需要指定的字段。比如，要指定标量资源，资源Type必须是SCALAR，scalar字段必须包含值。

最后，mesos.proto也通常是确定并且孵化Mesos新特性的地方。还没有任何官方文档的时候，mesos.proto里就能够看到即将发布的特性的描述。比如，服务发现系统和持久

化卷特性在稳定的几个版本前就出现在 mesos.proto 里。用户能够在 mesos.proto 里发现 Mesos 即将发生的变化；但是，如果不忽略和自己手头任务不相关的消息和字段的话，mesos.proto 也会因为信息过多而让人非常困惑。

不通过 Mesos 管理

要构建可以工作的 Mesos 集群，需要一些应用程序，而其很难或者无法作为框架运行：

ZooKeeper

> Mesos 使用 ZooKeeper 提供集群的发现服务。ZooKeeper quorum 由其运行机器的 DNS 名称命名。因此，Mesos 集群的 ZooKeeper quorum 通常在三台或者五台特定机器上运行，这样能保证拥有稳定的名称。

数据库

> 持久化卷的支持是最近在 Mesos 0.23 版本里才添加的。因此，大部分数据库还没有成熟的 Mesos 框架。尽管如此，Mesos 上的托管数据库的工作还是在快速推进。从 2014 年开始，Twitter 投入开发了 Apache Cotton（之前称为 Mysos），负责运行并且管理 Mesos 上的 MySQL。Mesosphere 也和其合作伙伴一起产品化了 Mesos 上的 Cassandra 和 Kafka，Basho 发布了 Mesos 的 Riak 框架的 beta 版本，它包含了 Mesos 启用的高性能代理层。

监控

> Mesos 观察并且监控其框架，帮助开发人员构建高可用性并且可靠的系统。但是，正如罗马诗人 Jubenal 所言，“谁观察观察器呢？”Mesos 本身也需要监控。它收集即时使用数据，但是这些数据必须归档并且图形化来帮助用户理解随时间演进的集群行为。另外，Mesos 并不是构建来定位其运行所在机器的问题的。因此，一个健壮的生产环境使用的 Mesos 集群必须带有主机级别的监控，并且它必须归档和警报 Mesos 自身收集的数据。该领域的一种集成方案是 Satellite，能够自动停用有问题的 slave，基于自定义条件生成警报，并且将历史数据存储到 InfluxDB、ElasticSearch，以及很多其他的数据库里。

幸运的是，搭建监控系统和 ZooKeeper 集群所需要的工作量，比搭建所有准备在 Mesos 上运行的框架的独立版本的工作量要小得多。

本章小结

Mesos 集群包括协调整个集群的 master，以及代替框架执行任务的 slave。框架也参照了 Mesos 集群的 master/slave 结构，每个框架包括调度器和控制运行在 slave 上的执行器。

可以使用额外的键 / 值元数据、自定义资源类型，和角色等自定义 Mesos slave，这些方式都能帮助确保框架调度器能够选择运行其执行器的最佳 slave，并且帮助管理员确保 Mesos master 的分配行为。在 web UI 上可以浏览 master 和 slave 的状态，web UI 能够在集群内交叉引用数据。

执行器是实际在 slave 的容器内运行的进程。在 Mesos 里，执行器能够托管一个或者多个任务，这给予框架作者一定的灵活性来组织其应用程序。但是，大部分框架使用 CommandExecutor 在每个执行器里只运行一个任务（也就是说，每个容器一个任务）。

最后，虽然 Mesos 提供了很多功能，用户仍然必须部署一些 Mesos 之外的工具：特别是，ZooKeeper quorum 来启动整个集群、监控系统，以及可能需要的一些数据库。

在本章我们初步了解了 Mesos，下面我们一起学习如何将一个已有的 web 应用程序放置到 Mesos 上！

第3章

29

将已有应用程序迁移到Mesos上

本章介绍如何在 Mesos 上构建应用程序。我们并不是完全从头开始构建，而是一起探讨如何利用已有框架将遗留应用程序迁移到 Mesos 上。大部分应用程序可以归为两大类：响应请求的应用程序以及在特定时间点完成操作的应用程序。在 LAMP 技术栈里，这两种组件分别是 PHP 和 cron 作业。

本章首先讲述如何将一个已有的基于 HTTP 的应用程序从现有基础架构里迁移到 Mesos 上。要实现这一目的，就需要利用 Mesos 的扩展性和弹性，最终构建出一个能够从常见故障里自动修复并且恢复的系统。除了改进应用程序本身的弹性，还需要改进应用程序组件之间的隔离性。这样做有助于达到更好的服务质量，而无须费力地直接在虚拟机上构建这些组件。这里将使用 Marathon，一个流行的 Mesos 框架，来托管基于 HTTP 的应用程序。

之后，会以 Chronos 作为学习案例，介绍如何使用 Marathon 为其他框架添加高可用和高可靠性。Chronos 让用户可以按照指定的间隔时间运行程序——可以用它计划设定每隔 15 分钟就启动一次夜间数据生成作业或者报告。之后我们还会介绍一些建议，关于如何高效且可维护地使用 Chronos。

最后简单介绍 Marathon 的一些替代方案，Singularity 和 Aurora，它们分别由 HubSpot 和 Twitter 公司开发。

将 Web 应用程序迁移到 Mesos 上

几乎每个公司都会有 web 应用程序。但是，无论是用 Ruby on Rails、Scala Play!，还是 Python 的 Flask 来编写，保证这些 web 应用程序的部署可靠、可扩展，以及可用性高，一直都是一个巨大的挑战。本节会介绍 Marathon——一个由 Mesosphere 开发的很易用

30

的平台即服务（PaaS）产品——及其如何和其他工具集成，比如 HAProxy。通过这样的学习，可以体会到之前所述架构的优势所在：

1. 所有后台进程都能够运行在任意机器上，Mesos 框架会在后台已有实例失败时，自动启动新实例。
2. 在完全不同的容器内托管静态资产、快速响应的端点、危险端点[1]，以及 API，来改进隔离性。
3. 更容易部署新版本或者回滚，在几秒钟之内就可以完成。
4. 将所有这些整合在一起，允许应用程序基于需求自动扩展。

本质上，Marathon 允许用户指定命令行或者 Docker 镜像、CPU 个数、内存总量，以及实例数，并且它会使用请求的 CPU 和内存数量，在容器内启动该命令行或者镜像的指定数量的实例。本节会研究这些基础配置，并且学习 Marathon 和负载均衡器集成的几种方式，从而将服务暴露给内部和外部世界。

搭建 Marathon

显然，要使用 Marathon，需要先下载。Mesosphere 网站上有为 Ubuntu、Debian、Red Hat 和 CentOS 预打包的 Marathon 版本。当然，用户也可以自己构建，从 Mesosphere 网站上下载最新的 tarball，或者从 GitHub 上克隆下代码。Marathon 用 Scala 编写，因此用户只需要运行 `sbt assembly` 就可以完成构建。从这里开始，本书假定已经完成了 Marathon 的安装。

Marathon 是否作为高可用性应用程序运行并不重要。实际上，在高可用配置下运行 Marathon 所需的所有事情仅仅是确保启动了两个或者三个 Marathon 实例，并且这些实例共享相同的 `--zk` 命令行参数。还必须确保其他参数也一致，比如—`master`、`--framework-name` 以及侦听 HTTP 请求的端口，否则就会看到令人困惑的行为。当以
31
这样的模式运行 Marathon 时，Marathon 的所有实例都可以正确地响应请求，并且它们会神奇地同步状态，因此用户完全不需要担心这些。通常通过给 Marathon 实例指派一个轮询（round-robin）的 DNS 名称，或者将其放在负载均衡器之后，就可以完成搭建了。表 3-1 列出了命令行参数相关的更多信息。

1 这里指的是该端口可能会在数据库上运行查询，或者构造一个外部服务调用，而用户无法预测该调用的完成需要多长时间，或者是否会无限制地消费资源。

表3-1 Marathon部分命令行参数

参数	功能
`--master` <instance>	Mesos master的URL。通常格式为`zk://host1:port,host2:port,host3:port/mesos`
`--framework-name` <name>	允许用户自定义Mesos里的Marathon的实例名称，在有多个Marathon实例运行时这很有用
`--zk` <instance>	保存了Marathon状态的ZooKeeper的实例集合。通常格式为`zk://host1:port,host2:port,host3:port/marathon`
`--https-port` <port>	启用Marathon的HTTPS访问。细节详见下面框中文字

保护 Marathon 和 Chronos

这里介绍如何通过在改动Marathon和Chronos之前强制要求认证来保护Marathon和Chronos，并且确保所有API通信都受SSL保护。虽然很多公司都是在其受信网络里运行Marathon和Chronos，但还是有一些公司需要应用访问限制来确保只有授权用户才能查看或者更改运行着的作业。注意这并不会保护Marathon或Chronos和Mesos之间的连接，要达到这个目的，用户需要为所选择的Mesos用户认证方案配置`--mesos_authentication_principal`和`--mesos_authentication_secret_file`。这样做能够有效地防止有人不小心删除或者改变生产环境上至关重要的服务，也能防御想进行恶意破坏的员工。

32

所有这些安全设置都可以通过环境变量或者命令行参数配置。最好使用环境变量，因为任何登录运行着Marathon或Chronos机器的人都可以通过运行`ps -f`看到命令行参数里的密码。让启动应用程序的脚本在启动之前`export`所有敏感的环境变量，这样可以保护环境变量。同时设置启动器脚本的权限控制，让机器的大部分用户不可读，从而保护所使用的密码。

Chronos和Marathon都是基于相同的名为Chaos的HTTP/REST服务库而构建。Chaos目前仅仅支持HTTP基础认证，即用户名/密码认证。要设置`username:password`，用户需要遵守这样的格式（由冒号分隔）。可以通过如下参数将其传递给命令行：

```
--ssl_keystore_password password
```

或者通过环境变量：

```
MESOSPHERE_KEYSTORE_PASS=password
```

这时，你很可能会问："为什么 Java 总是想以不同的方式完成每件事情呢？为什么它不能直接兼容常规证书呢？"幸运的是，将标准 PEM 格式的 PKCS12 证书（这是大多数证书发行机构使用的类型）转换成 keystore 很容易。要达到这一目的，用户需要 key（mykey.pem）和证书（mycert.pem）。首先，运行如下命令生成一个组合文件（mykeycert.pem）：

```
cat mykey.pem mycert.pem > mykeycert.pem
```

然后，使用 openssl 创建包含证书的 keystore（mykeystore.pkcs12）。很奇怪的是，keystore 要求其存储的所有证书都要有"别名"。本书的证书别名为 myalias。最终命令如下：

```
openssl pkcs12 -export -in mykeycert.pem -out mykeystore.pkcs12 \
-name myalias -noiter -nomaciter
```

成功啦！现在就可以使用任何标准颁发的证书来保护 Marathon 和 Chronos 了。

完成上述配置之后，一定要确保禁用 HTTP 访问，否则就相当于在前门上锁，但是后门敞开了！可以通过命令行参数 `--disable_http` 禁用 Marathon 和 Chronos 的 HTTP 访问。

33

使用 Marathon

Marathon 完全通过其 HTTP API 来控制。当 Marathon 运行着时，让我们尝试启动一个应用程序。

这里会运行一个简单的应用程序：著名的 Python `SimpleHTTPServer`。这一应用程序在启动时会简单暴露其目录下的文件。本书示例以此为基础，因为几乎所有人都安装了 Python，并且默认可用。

如果在你的机器上运行如下命令：

```
python -m SimpleHTTPServer 8000
```

那么在浏览器里打开 localhost:8000，就能够看到启动服务器的目录下的文件列表。

一起看看如何在运行在 `marathon.example.com:8080` 的 Marathon 服务器上运行这一应

用程序。首先构造一个文件，包含示例 3-1 中所示的 Marathon 应用程序的 JSON 描述符。

示例3-1 SimpleHTTPServer JSON描述符

```
{
    "cmd": "python -m SimpleHTTPServer 31500", ❶
    "cpus": 0.5, ❷
    "mem": 50, ❸
    "instances": 1, ❹
    "id": "my-first-app", ❺
    "ports": [31500], ❻
    "requirePorts": true ❼
}
```

❶ “cmd”指定将要启动应用程序的命令行。

❷ “cpus”指定应用程序容器应该占用的 CPU 数。该数值可以是分数。

❸ “mem”指定应用程序容器应该占用的内存 MB 数。

❹ “instances”指定在集群里应该启动的应用程序拷贝数。本演示中，仅仅生成一个实例，但是典型的应用程序可以生成 2 到 2000 个实例。

❺ “id”是将来通过 API 如何指代该应用程序。对于每个应用程序而言该值必须唯一。

❻ “ports”是应用程序要求的端口数组。本例只需要一个端口。之后会讲解如何动态分配和绑定到端口。 34

❼ 因为显式指定想要分配的端口，所以必须告诉 Marathon，否则它会默认使用自动端口分配。

现在假定上述数据已经存储在名为 *my-first-app.json* 的文件里。为了启动应用程序，使用 HTTP POST 将数据发送给运行在 marathon.example.com:8080 的 Marathon 服务器：

```
curl -X POST -H 'Content-Type: application/json' \
    marathon.example.com:8080/v2/apps --data @my-first-app.json
```

会得到类似如下的响应：

```
{
   // 该部分包含指定的参数
   "id" : "/my-first-app",
   "instances" : 1,
   "cmd" : "python -m SimpleHTTPServer 31500",
   "ports" : [ 31500 ],
   "requirePorts": true,
   "mem" : 50,
   "cpus" : 0.5,
```

```
    // 该部分包含自动指定值的参数
    "backoffFactor" : 1.15,
    "maxLaunchDelaySeconds" : 3600,
    "upgradeStrategy" : {
       "minimumHealthCapacity" : 1,
       "maximumOverCapacity" : 1
    },
    "version" : "2015-06-04T18:26:18.834Z",
    "deployments" : [
       {
          "id" : "54e5fdf8-8a81-4f95-805f-b9ecc9293095"
       }
    ],
    "backoffSeconds" : 1,
    "disk" : 0,
    "tasksRunning" : 0,
    "tasksHealthy" : 0,
    "tasks" : [],
    "tasksStaged" : 0,

    // 该部分包含未指定的参数
    "executor" : "",
    "storeUrls" : [],
    "dependencies" : [],
    "args" : null,
    "healthChecks" : [],
    "uris" : [],
    "env" : {},
    "tasksUnhealthy" : 0,
    "user" : null,
    "requirePorts" : true,
    "container" : null,
    "constraints" : [],
    "labels" : {},
}
```

35

这表明应用程序已经正常启动了。

查询 my-first-app（这个应用程序的名称）相关的信息，就可以得到这个运行着的应用程序的所有信息：

```
curl marathon.example.com:8080/v2/apps/my-first-app | python -m json.tool
```

该查询返回的数据和创建应用程序返回的数据几乎完全一致。不同之处在于这里应用程序已经开始运行了一段时间，因此可以看到运行任务相关的信息如下：

```
// 省略无关代码
    "tasks": [
        {
            "appId": "/my-first-app",
            "host": "10.141.141.10", ❶
            "id": "my-first-app.7345b7b5-0ae7-11e5-b3a7-56847afe9799", ❷
            "ports": [ 31500 ], ❸
            "stagedAt": "2015-06-04T18:28:09.235Z", ❹
            "startedAt": "2015-06-04T18:28:09.404Z", ❺
            "version": "2015-06-04T18:28:05.214Z" ❻
        }
    ],
    "tasksHealthy" : 1, ❼
// 省略无关代码
```

❶ 该值告诉用户运行该任务的主机的 DNS 名称和 IP 地址。

❷ 任务的 id 是一个 Mesos 结构。可以使用 Mesos master 的 UI 来检查任务，比如，可以查看其 `stdout` 和 `stderr`。

❸ 是否选择应用程序运行的端口，或者允许 Marathon 代替用户做出选择（细节见示例 3-2），可以找到任务实际预留的端口。

❹ 任务的 staging 时间是将该任务提交给某个 Mesos offer 的时间。

❺ 任务的开始时间是 Mesos 实际启动任务的时间。

❻ 每次更新 Marathon 描述符时，都会标记成一个新版本（见最初 `POST` 的响应）。每个任务都会在其注释里标明启动版本。 〈36

❼ 这里给出了应用程序里当前健康任务的数量概览。

可以进一步查看一些应用程序相关的更为丰富的信息。首先，可以看到创建应用程序时指定的所有配置参数。还可以看到很多其他设置，其中大部分本书都会介绍。还有一个很有意思的字段是“`tasks`”，是应用程序的所有实际运行实例的数组。每个任务都表示为 JSON 对象，带有一些有用的字段：

- “`host`”和“`ports`”是分配给任务的实际主机和端口。这些字段是了解应用程序实际上在哪里运行的关键，从而用户可以连接上去。
- “`id`”是 Mesos `TaskID`。将每个任务唯一的 UUID 添加到应用程序 ID 上而构造出来的，从而可以通过 Mesos UI 里的 Mesos CLI 工具很便捷地实现定位。
- “`stagedAt`”和“`startedAt`”告诉用户什么时候任务上发生了生命周期事件。一旦 Marathon 实际请求 Mesos 在特定 offer 上启动任务，任务就变成 staged；一旦任务开始运行就变成 started。

Marathon REST API 即将发生的改动

本书的 REST API 示例基于 Marathon 0.9 版本。将来，Marathon 会对结果进行过滤，这样请求就无须返回兆比特的 JSON 数据。当这一功能实现时，用户需要制定额外的可重复的 URL 参数 embed。也就是说，它会取代现在的 `/v2/apps/my-first-app`，变为使用请求 `/v2/apps/my-first-app?embed=app.counts&embed=app.tasks` 来获得上述示例的数据。Marathon REST API 文档里介绍了 embed 的所有参数。

通常，用户不想硬编码应用程序的端口：毕竟，如果这么做了，那么每个 slave 上就只能运行该应用程序的一个实例。Marathon 可以替用户指定端口：将应用程序所要求的端口简单设置为 0，并且将分配的端口记录在环境变量 `$PORT0`、`$PORT1` 等里。利用这一功能改动之前的配置，就会如示例 3-2 所示。

37 示例3-2 SimpleHTTPServer JSON描述符，使用动态选择的端口

```
{
    "cmd": "./python -m SimpleHTTPServer $PORT0", ❶
    "cpus": 0.5,
    "mem": 50,
    "instances": 1,
    "id": "my-first-app",
    "ports": [0], ❷
    "uris": [
        "http://fetchable/uri/with/python.tar.gz" ❸
    ]
}
```

❶ 注意使用 `$PORT0` 访问环境变量，其包含分配给“ports”数组的首个索引的端口。

❷ 设置所需端口为 0，意味着让 Marathon 做选择。

❸ 还提供待获取的 URI 列表，在运行命令行之前解压缩到容器内。支持的 URI 命名方案详见“配置进程环境”部分。

一些想要运行在 Marathon 上的应用程序不接受命令行传入的端口。不用担心！用户可以很容易地将 `$PORT0` 环境变量传递进配置文件。假定应用程序配置文件设置为和应用程序一起下载的 URL。现在，想要使用 sed 将配置文件里的某个特定字符串替换为必需的环境变量，可能是 `$PORT0` 或者 `$MESOS_DIRECTORY`：

```
sed -e 's#@MESOSDIR@#'"$MESOS_DIRECTORY"'#' config.templ > config ❶
sed -e 's#@PORT@#'"$PORT0"'#' config2.templ > config2 ❷
```

❶ 这里将字符串 `@MESOSDIR@` 替换为任务的沙箱目录。将配置文件里的字符串用 @ 符号包围，因为 @ 符号几乎不会出现在配置语言里（当然，其实可以使用任意的特别字符

串）。另外，在 sed 命令行里使用单引号包含“常量”，用双引号包含环境变量，从而确保 shell 能够正确执行该命令。最后，在 sed 命令里使用 # 代替 /，这样就可以成功放置本身包含斜杠的环境变量。

❷ 这里将索引为 0 的端口放置到配置文件里。实际上，如果想在一个配置文件里放置多个变量，可以向 sed 传递多个 -e 参数。

之后介绍使用这些变量的特性时会再介绍其他字段。

38

扩展应用程序

本节探讨如何将 HTTP 服务器扩展到 5 个实例。准备好了吗？

```
curl -X PUT -H 'Content-Type: application/json' \
    marathon.example.com:8080/v2/apps --data '{"instances": 5}'
```

当然，没有人会真的需要将这个没什么用的应用程序启动 5 个实例，因此为什么不缩小实例数呢？

```
curl -X PUT -H 'Content-Type: application/json' \
    marathon.example.com:8080/v2/apps --data '{"instances": 1}'
```

真幸运，伸缩相当容易！

使用位置约束

Marathon 可以约束在何处启动应用程序。这些约束可以是 slave 的主机名或者任意 slave 属性。约束放在数组里提供给应用程序，每个约束本身是包含两个或三个元素的数组，取决于是否需要传入参数。下面是可用的约束及其使用方法：

GROUP_BY

该运算符确保应用程序的实例在带有特定属性的节点上均匀分布。可以借助该值在主机或者机架（假定机架信息记录在 slave 的 rack 属性里，示例 3-4）间均匀分发应用程序（见示例 3-3）。

示例3-3 主机间均匀分布

```
{
   // 省略无关代码
   "constraints": [["hostname", "GROUP_BY"]]
}
```

示例3-4 机架间均匀分布

```
{
    // 省略无关代码
    "constraints": [["rack", "GROUP_BY"]]
}
```

UNIQUE

该运算符确保应用程序的每个实例都运行在UNIQUE约束值不相同的机器上。该值类似于GROUP_BY，除了一点：当缺少足够的具有不同该属性值的机器时，应用程序的完整部署就会失败，而不会在某些slave上运行多个实例。通常，GROUP_BY是保证可用性的最佳方案，因为其通常优先在单台slave上运行多个任务。示例3-5展示如何使用UNIQUE确保集群内的每个slave上不会有多于一个实例运行。

39

示例3-5 每台主机上最多运行一个实例

```
{
    // 省略无关代码
    "constraints": [["hostname", "UNIQUE"]]
}
```

CLUSTER

该运算符允许用户仅仅在某个属性值为特定值的slave上运行应用程序。如果某个slave带有特别的硬件配置，或者如果用户想限制应用程序在某个机架上执行，这时该运算符就很有用。但是，LIKE更强大，并且通常更实用。比如，假定用户的数据中心是ARM和x86处理器的混合，并且在slave上设置了属性cpu_arch来帮助区分机器的架构。示例3-6展示如何确保仅仅在x86处理器的机器上运行应用程序。

示例3-6 仅仅运行在x86上

```
{
    // 省略无关代码
    "constraints": [["cpu_arch", "CLUSTER", "x86"]] ❶
}
```

❶ 注意CLUSTER运算符带了一个参数。

LIKE

该运算符确保应用程序仅仅运行在拥有某个特定属性的slave上，并且该属性值能够匹配所提供的正则表达式。该运算符的使用方法和CLUSTER类似，但是更为灵活，因为它能够匹配很多值。比如，假定集群运行在Amazon上，并且所有slave设置了instance_type属性。示例3-7展示了如何限制应用程序仅仅运行在最大的C4

计算优化的机器上。

示例3-7 仅仅运行在c4.4xlarge和c4.8xlarge机器上

40

```
{
    // 省略无关代码
    "constraints": [["cpu_arch", "LIKE", "c4.[48]xlarge"]] ❶
}
```

❶ LIKE 的参数是正则表达式。

UNLIKE

UNLIKE 是 LIKE 的补充：它帮助避免在某些特定的 slave 上运行。可能用户不想在 DMZ 内的机器上运行后台程序（比如处理用户海量信息的程序），因为 DMZ 处在不受保护的网络区域。在 DMZ 内的机器上将 dmz 属性设置为 true，就可以避免在这些机器上运行应用程序，如示例 3-8 所示。

示例3-8 不在DMZ内运行

```
{
    // 省略无关代码
    "constraints": [["dmz", "UNLIKE", "true"]] ❶
}
```

❶ UNLIKE 的参数也可以是正则表达式。

位置约束还可以进行组合。比如，确保应用程序不在 DMZ 内运行，并且在其他机器上均匀分布，如示例 3-9 所示。

示例3-9 组合的约束

```
// 省略无关代码
"constraints": [["dmz", "UNLIKE", "true"],
                ["hostname", "GROUP_BY"]]
```

组合约束帮助用户精确控制应用程序在集群内的运行位置。

运行容器化的应用程序

Marathon 很好地支持了 Docker 容器。如果应用程序已经容器化了，就很容易在 Marathon 上运行。

为了支持 Docker，首先需要确保 Mesos 集群已经启用了 Docker 支持。要达到这一目的，需要确保已经启用 Docker 容器机，并且增加执行器的超时时间（详见“使用 Docker”部分）。

41

本节仅仅在配置应用程序的 JSON 里添加 Docker 配置。示例 3-10 展示了使用 Docker 时用户可用的参数。注意该 JSON 配置文件只是完整文件的一部分，完整配置必须包含示例 3-1 所示的参数。

示例3-10 Marathon Docker JSON配置

```
{
    // 省略无关代码
    "container": {
        "type": "DOCKER", ❶
        "docker": {
            "image": "group/image", ❷
            "network": "HOST" ❸
        }
    }
}
```

❶ 因为 Mesos 会继续添加新的容器类型，比如 Rocket 或 KVM 容器，所以大部分容器配置并非特定于 Docker。该示例里必须提供 Docker 特定的配置，因为这里用的是 Docker 容器。

❷ 这是最重要的部分：指定容器镜像在哪里。[1]

❸ 将在主机网络模式下运行 Docker，Docker 会直接使用 slave 的网络资源。

一旦 Docker 容器启动了，它就能访问 Mesos 沙箱目录，该目录存储在环境变量 $MESOS_SANDBOX 里。

挂载主机卷

使用 Docker 时，用户通常需要挂载一些主机上可用的卷，可能是一些全局性分布式配置文件、数据文件，也可能是不会被自动垃圾回收的数据目录。示例 3-11 展示了如何将这些卷添加到 JSON 配置里。

42

示例3-11 Mount主机卷

```
{
    "container": {
        "type": "DOCKER",
        "docker": {
            "image": "group/image",
            "network": "HOST"
        },
```

1 如果想要使用私有 Docker 存储库，可以在应用程序描述符里将 .dockercfg 添加到“uris”字段，详情见“使用 Docker”小节的注释“高级 Docker 配置”。

```
        "volumes": [ ❶
            { ❷
                "containerPath": "/var/hostlib",
                "hostPath": "/usr/lib",
                "mode": "RO" ❸
            },
            { ❹
                "containerPath": "/var/scratch",
                "hostPath": "/mount/ssd",
                "mode": "RW"
            }
        ]
    }
}
```

❶ 注意“volumes”并不是 Docker 特有的，并且这里可以指定一个数组。

❷ 卷允许用户从 Docker 容器内访问宿主机器的 /usr/lib 目录。这在用户想要访问主机的 libmesos.so 时很有用，就无须持续更新 Docker 镜像了。

❸ 可以以只读（RO）或者只写（RW）模式挂载卷。

❹ 该卷给了用户一些 SSD 挂载的空白空间，假定该 SSD 挂载在主机的 /mount/ssd 上。

将主机卷挂载到 Docker 容器内似乎是在 Marathon 上托管数据库的一种好方法，这样，可以向容器提供持久化的主机存储。不过，即使可以向 Docker 上挂载卷，这么做其实也很困难（至少，只有等到 Mesos 原生支持卷时情况可能才会有所改善）。首先，沟通哪台机器有可用磁盘这件事就十分枯燥。为了达到这一目的，需要使用位置约束来确保如下几个方面：

1. 有空闲卷的每个 slave 需要设置特别的属性，表明其有空白卷，这样才可以使用 CLUSTER 运算符来确保使用这些 slave。

2. 应用程序的 UNIQUE 运算符设置为主机名，这样每台 slave 上仅仅会启动应用程序的一个实例（因为每台机器仅有一个特定的加载点）。 <43>

3. 每个 slave 的主机卷加载在相同路径上（很幸运的是，这是最容易办到的）。

如果能够完成上述任务，那么就能够在 Marathon 上运行某种类型的数据库。但是不幸的是，无法支持扩展或者自动恢复，因为每个 slave 上有太多必需的特别配置要求。不过，随着 Mesos 对持久化磁盘支持的完善，会涌现出越来越多的处理启动和配置数据库的专业框架，也许不久之后这里的困难就不复存在了。

健康检查

Marathon 会持续跟踪应用程序是否健康。通过 REST API 暴露这些信息，从而可以很容易地使用 Marathon 中央化管理所有应用程序的健康检查。当用户查询 REST API 意图得到特定应用程序的信息时，其所有任务都会包含最新的健康检查结果。默认来说，Marathon 假定处于 RUNNING 状态的任务是健康的。当然，某个应用程序已经开始执行并不意味着它实际上就是健康的。为了改进应用程序的无效健康数据，用户可以指定任务必须满足的三种额外类型的健康检查：基于命令、HTTP 和 TCP。

示例3-12 常见健康检查字段

```
{
    "gracePeriodSeconds": 300, ❶
    "intervalSeconds": 60, ❷
    "maxConsecutiveFailures": 3, ❸
    "timeoutSeconds": 30 ❹
}
```

❶ 该宽限期是任务启动自身所需花费的给定时间。在这段时间用完之前，不会运行针对该任务的健康检查。该参数是可选的，默认值是 5 分钟的宽限期。

❷ 间隔是两次健康检查运行之间的时间。该参数是可选的，默认值是每次检查间隔 1 分钟。

44

❸ 连续失败的最大次数，决定多少次失败的健康检查之后 Marathon 会强制“杀死”该任务，从而让这个任务能在其他地方重启。如果该值设置为 0，那么健康检查的失败永远都不会导致“杀死”任务。该参数是可选的，默认三次健康检查失败后任务就会被“杀死”。

❹ 有时候，健康检查可能会挂起。Marathon 并不会简单地忽略这次检查，而是当检查在这里给定的秒数之内没有完成时，Marathon 会认定缓慢或者挂起的健康检查失败了。该参数是可选的，默认健康检查的运行时间为 20 秒。

本节介绍健康检查三种类型之中的两种：HTTP 和 TCP。命令式健康检查还相对比较新，并且因为它有一些限制，所以建议等一段时间该功能更为稳定之后再使用。不过要记住，这里的 HTTP 和 TCP 健康检查目前是由 Marathon master 执行的，这意味着它们还无法扩展到上千的任务里。[1]

HTTP 检查

HTTP 健康检查验证在某个特定的路由发送 GET 请求是否会得到成功的 HTTP 状态码。成功状态码可以是真的成功，或者重定向。因此，响应码必须在 200-399 的范围内，包

1 Mesos 以后的版本里会增加横向扩展的功能。可以在 https://issues.apache.org/jira 处查看 MESOS-2533 和 MESOS-3567，看到是否已经添加这些特性。

含 200 和 399。要配置 HTTP 健康检查，需要两个必需字段和一个可选字段，如示例 3-13 所示。

示例3-13 HTTP健康检查字段

```
{
    "protocol": "HTTP", ❶
    "path": "/healthcheck", ❷
    "portIndex": 0 ❸
}
```

❶ 必须在“protocol”字段指定这是一次 HTTP 健康检查。

❷ 必须指定健康检查的路径。本示例尝试向 `http://myserver.example.com:8888/healthcheck` 发送 GET 请求，这里 `myserver.example.com:8888` 是 Marathon 在其上启动任务的主机和端口。

❸ 可以为健康检查选择性指定“portIndex”。默认端口索引为 0。

为什么健康检查使用“portIndex”而不是“port”？

可能大家会很困惑健康检查规范里的“portIndex”是什么，为什么不简单指定实际端口呢。记住 Marathon 能够（最佳实践也是这么推荐的）选择应用程序所绑定的端口。所以，用户在配置健康检查时实际并不知道应用程序会运行在哪个端口上。但是应用程序使用的所有端口都指定在应用程序描述符的“ports”数组里。因此，用户可以简单指定“ports”数组里的哪个索引对应为想要在其上运行健康检查的端口！

TCP 检查

TCP 健康检查验证是否能够成功开启 TCP 链接到任务上。它们并不发送或者接受任何数据，而只是简单尝试开启 socket。要配置 TCP 健康检查，需要配置一个必需字段和一个可选字段，如示例 3-14 所示。

示例3-14 TCP健康检查字段

```
{
    "protocol": "TCP", ❶
    "portIndex": 0 ❷
}
```

❶ 必须在“protocol”字段指定这是一次 TCP 健康检查。

❷ 可以为健康检查选择性地指定“portIndex”。默认的端口索引为 0。

应用版本化和滚动升级

如果分布式可扩展的 PaaS 不能滚动升级，它还有什么用呢？一次升级指的是修改应用程序的任意时间点，通常通过发送 PUT 请求到 Marathon 上的 `/v2/apps/$appid` 处来完成升级。每次升级都会创建一个新版本，由升级所发生的时间戳命名。当查询 REST API 得到应用程序信息时，每个任务都会告知正在运行的是哪个版本。用户可以在 Marathon 的 `/v2/apps/$appid/versions` 路径处查询有哪些版本可用，还可以从 `/v2/apps/$appid/versions/$version` 路径得到该版本的详细信息。示例 3-15 展示了如何配置应用程序的升级行为。

示例3-15 应用程序升级

```
{
    "upgradeStrategy": {
        "minimumHealthCapacity": 0.5,
        "maximumOverCapacity": 0.1
    }
}
```

使用 Marathon，用户可以通过应用程序描述符的“`minimumHealthCapacity`”字段，指定在升级中有多少应用程序需要保持可用。“`minimumHealthCapacity`”可以设置为 0 到 1 之间的任何值。

如果设置为 `0`，当部署新版本的 Marathon 时，会首先“杀死”所有旧任务，然后启动新任务。

如果设置为 `0.6`，当部署新版本的 Marathon 时，会首先“杀死”40% 的旧任务，然后启动 60% 的新任务，然后“杀死”剩余的旧任务并且启动剩余的新任务。

如果设置为 `1`，当部署新版本的 Marathon 时，会首先启动所有新任务（这时应用程序使用的资源可能是常规运营时的两倍），然后关闭旧任务。

“`maximumOverCapacity`”设置提供了额外级别的安全性，这样应用程序不会在升级中瞬时消耗太多的资源。“`maximumOverCapacity`”可以设置为 0 到 1 之间的任何值。

如果设置为 `0`，当部署新版本的 Marathon 时，不会启动比实例的目标数量多的任务。

如果设置为 `0.6`，当部署新版本的 Marathon 时，永远不会超过实例目标数量的 160%。

如果设置为 `1`，当部署新版本的 Marathon 时，则会允许在“杀死”任何旧任务之前就启动所有新任务。

事件总线

当使用 Marathon 构建应用程序时，用户通常希望在各种事件发生时接收到通知，比如当某个应用程序有了新的部署，或者当其扩容或缩容时。这些通知能够随后用于重新配置代理和路由，做日志分析，并且生成报告。Marathon 有个内建特性，能够自动将所有内部事件 POST 到给定 URI 处。要使用该特性，仅仅需要向 Marathon 提供两个额外的命令行参数：

`--event_subscriber`

该参数必须传入值 `http_callback`，才能启用该子系统。现在还不支持其他方案。

`--http_endpoints`

该参数是由逗号分隔的向其发送 JSON 格式事件的目标 URI 列表。比如，有效参数类似于 `http://host1/foo,http://host2/bar`。

搭建 Marathon 上的 HAProxy

我们在上文中已经学习了 Marathon，并且一起体会到为什么说 Marathon 是应用程序的绝佳平台。至此，应用程序已经能够在 Marathon 上运行了，但是还有一个问题：外部世界是如何真正连接到应用程序上去的呢？一些类型的应用程序，比如 Celery 或者 Resque worker，不需要任何额外的配置：它们通过共享数据库通信。其他类型的应用程序，比如 HTTP 服务器和 Redis 实例，需要确保其很容易被发现。在 Mesos 集群里解决这个问题的最流行方式是在静态，非 Mesos 管理的主机上运行代理，并且自动更新这些代理，将其指向应用程序运行着的实例。这样，应用程序在已知主机和端口（也就是代理）上持续保持可用，同时仍然可以在后台动态扩展其能力。另外，每个代理主机都能够服务很多应用程序，每个应用程序放在不同的端口上，这样少量的代理主机就能够运行上百个后台程序。通常来说，如果代理提供 SSL 终止功能，它会受所有活跃连接的总带宽，或者 CPU 使用总量的限制。[1]

有两种代理无疑是现代应用程序栈使用最多的方案：HAProxy 和 Nginx。HAProxy 是功能很有限的代理：它能够代理 HTTP 和 TCP 连接，执行基本的健康检查，并且提供 SSL 终止功能。它是为稳定性和性能而构建的：HAProxy 在 13 年里都没有出现任何崩溃或死锁问题。这也正是现在 HAProxy 在 Mesos 用户中如此受欢迎的原因。

与之对比，Nginx 拥有很多特性。除了作为代理，它还能够运行自定义的，用 Lua、

1 SSL 终止功能是当客户端通过安全 SSL 连接和代理通信时，代理会处理 SSL，这样后台无须在加解密上浪费 CPU 资源，甚至无须支持加解密。

JavaScript，或者基于 JVM 的语言编写的用户代码。这些代码可能会影响代理的操作，甚至能够直接作出响应。很多系统实际上使用这两种代理：客户端连接到 HAProxy 实例
48 上，随后 HAProxy 将请求转发给 Nginx 实例。Nginx 实例随后要么直接回复，要么发送到后台处理请求的服务器上。

本节仅仅讨论 HAProxy，但是如果需要使用 Nginx 所提供的额外功能，那么要知道 Nginx 也完全可以按照本节介绍的方式使用。

haproxy-marathon-bridge

在 Marathon 存储库里，有一个名为 *bin/haproxymarathon-bridge* 的脚本。该脚本意图为 Marathon 生成一份 HAProxy 的配置文件。不幸的是，它并不允许用户指定 Marathon 上服务的哪个端口会绑定到 HAProxy 主机上。因此，该脚本基本没什么用。

下文介绍搭建 Mesos 代理的最为流行的配置。

Bamboo

Bamboo 是一个带有 web 接口的 HAProxy 配置后台程序。它提供 UI 来查看以及管理 HAProxy 规则的当前状态。Bamboo 的核心功能是监控 Marathon 上应用程序的更改，并且保证 HAProxy 实例的后台程序使用的是最新版本。Bamboo 还提供一些有用的特性，所有这些都通过 web 接口和 HTTP REST API 暴露出来。

最重要的是，Bamboo 支持为每个 Marathon 应用程序使用 HAProxy ACL 规则。ACL 规则能够基于其 URL、cookie、头信息和其他因素来路由请求。比如，可以使用 ACL 规则将路径由 `/static` 开头的请求路由到处理静态内容的后台程序，同时将所有其他请求路由到应用程序服务器。ACL 规则相当强大，因为它们允许用户将自己的应用程序分隔成独立的组件或者服务，从而分别扩展这些组件，同时暴露给客户端的 URL 仍然是唯一的。使用 ACL 规则，用户可以选择将匿名访客路由到某个服务器池，登录用户路由到另一个服务器池，并且将静态数据请求路由到第三个服务器池，从而可以单独扩展这些服务器池的能力。

Bamboo 还有一个强大的模板系统，基于 GO 语言的 `text/template` 包。该模板系统很灵活，但是和更简单更流行的系统，比如 mustache 模板相比，其学习曲线更陡峭一些。

所有代理配置信息都存储在 ZooKeeper 里，这样可以通过保证 Bamboo 所有实例的同步来简化配置管理。

49 Bamboo 使用 Marathon 事件总线来发现后台配置的变更，这意味着 HAProxy 配置在

Marathon 上的变更通常仅仅会延时数百微秒。

另外，Bamboo 能够和 StatsD 集成，从而报告配置事件发生的时间。

Bamboo 为 Marathon 应用程序的精确 HAProxy 配置提供了完备的方案。

微服务的 HAProxy

如今，微服务是一种非常流行的架构模式。Mesos 和 Marathon 是构建基于微服务的应用程序的坚实基础：用户可以在 Marathon 上托管所有的服务。不幸的是，服务发现是微服务部署时必须要解决的一个重要问题。下文介绍如何使用 Mesos 和 Marathon 创建类似于 SmartStack 的系统。

这里需要考虑两个问题：如何做服务发现？标准方案的问题是什么？标准方案包括：

DNS

> 可以选择使用轮询 DNS 来给所有后台服务器相同的 DNS 名称。但是这里会遇到两个问题：通常需要花费至少几秒钟才能在整个基础架构上完成 DNS 的变更，而且一些应用程序会永远缓存 DNS 解析（意味着这些应用程序永远不会知道 DNS 的变更）。另一个问题是，除非用户编写自定义的使用 SRV 记录的客户端库函数，否则 DNS 并不能提供简单的方案在同一个服务器上运行多个应用程序，并且让这些应用程序使用不同或者随机分配的端口。

中央化的负载均衡

> 这一点类似于“Bamboo”小节所讨论的问题。对于一些应用程序而言，这的确是个伟大的设计，但是，该方案要求全局可访问的负载均衡器池。这样的中央化让应用程序的安全隔离工作显得很枯燥：为了隔离应用程序，用户必须为特定的隔离需求配置 Nginx 或者 HAProxy 规则。[1]

应用内发现

> Twitter 和 Google 这样的公司使用特别定制的服务请求和发现层，使用的分别是 Finagile 和 Stubby。这些 RPC 层将服务发现直接集成进应用程序内，在需要找到可以发送请求的服务器时，查询 ZooKeeper 或 Chubby（Google 内部的类似 ZooKeeper 的系统）。如果可以在环境里这么实现，那么使用特别定制的服务发现层能够给路由、负载均衡和可靠性方面提供最大的灵活度，因为用户可以控制该系统所有方面的细节。不幸的是，如果需要和用多种语言编写的应用程序集成，就会

50

1 另一种方案，DNS 和应用内发现，将通信直接委托给每个应用程序，这样就无须为隔离而配置任何中央系统。相反，每个应用程序能够在和其他应用程序完全隔离的情况下，处理认证授权。

变得很麻烦并且很费时间，因为需要为每种语言开发平行的发现系统。此外，如果需要和已有的，没有遵守内建服务发现机制开发的应用程序集成的话，该方案则完全不可行。

上述这些方案的缺点导致很多工程师选择了第四种模式：联合中央化负载均衡的易于集成的特点和去中央化方案的容错特点的方案。使用该方案，用户会在数据中心里的每台单独机器上运行一个 HAProxy 的副本，然后给每个应用程序分配一个端口（不是主机名）。最后，将所有 HAProxy 实例都分别指向其后台应用程序的端口，无论这些后台程序运行在集群的什么位置。

这一方案有两大优势：

易于搭建

用户可以在集群内每台机器上使用 Bamboo 或者一个简单的 cron 作业来保持 HAProxy 本地实例和 Marathon 的同步。即使基础架构的某些部分暂时不可用，本地 HAProxy 也会继续将路由流量直接发送给后台程序。也就是说，即使 Marathon 崩溃了，也不会影响到应用程序的运营。

易于使用

每个应用程序都可以简单地连接到 `localhost:$PORT` 和其他微服务通信，这里 `$PORT` 是分配给微服务的端口。不需要为应用程序编写任何特别的或者自定义的代码，因为所有需要做的事情就是记住哪个端口对应哪个应用程序而已。

该方案也有一些缺陷。其中之一是用户需要仔细维护服务到端口的映射。如果用户尝试全局存储这样的映射，让应用程序在启动时读取映射信息，那么就必须再次为每种语言编写自定义代码。另一方面，如果本身硬编码了端口，就等于自挖陷阱，出现问题时调试会非常困难。

另一个缺陷是代理间缺少协调。即使发生局部故障，系统还会继续工作，因为每台机器都是独立操作的。也正是由于每台机器的独立操作，可能某些 HAProxy 会路由到相同的后台程序上，从而加重这个后台程序的负载，造成层级故障。随着集群内节点数的增加，经常会看到请求时间的方差也会随之增加。

51

如今，服务映射到端口，并且每个客户端都连接到 `localhost` 的 HAProxy 上的方案似乎已经成为管理 Mesos 集群上服务发现最为流行的方式了。

在 Marathon 上运行 Mesos 框架

Mesos 集群里的常见方案是在 Marathon 上运行集群的 Mesos 框架。但是 Marathon 本身就是一种 Mesos 的框架！那么在 Marathon 上运行 Mesos 框架意味着什么呢？不用考虑如何将每种框架的调度器部署到特定的主机上并且处理这些主机的故障，Marathon 能够确保框架的调度器总是在集群里的某处运行着。这样大幅简化了在高可用配置里部署新框架的复杂度。

如何给由 Marathon 运行的框架分配资源？

Mesos 的新用户通常都会问，“如果我在 Marathon 上运行一个框架，这会如何影响资源的分配呢？”实际上，虽然框架的调度器是由 Marathon 运行的，但是调度器仍然必须直接连接到 Mesos 上，并且调度器会以和在特定机器上直接运行它们完全一样的方式接受资源。当然，在 Marathon 上运行调度器的优势是任意 Mesos slave 上都能够运行调度器，并且当 slave 发生故障时，Marathon 能够自动重启调度器。

Chronos 是什么

Chronos 是一种 Mesos 框架，提供高可用性，分布式基于时间的作业调度器，比如 cron。借助于 Chronos，用户能够启动命令行程序（可能是从 URI 处下载的）或者 Docker 容器。和 Marathon 一样，Chronos 有 web UI，以及易于编程的 REST API。虽然 Chronos 端点和 Marathon 端点的接口有细微的不同，但是本书不会加以详细介绍，因为在线文档非常详尽。Chronos 提供了调度作业的四大核心特性：

间隔调度

Chronos 使用标准 ISO 8601 注释指定重复间隔。通过 web UI 或者 REST API，用户可以指定重新运行作业的频率。这使得 Chronos 很适合每隔几分钟运行一次快速作业，或者运行夜间的数据生成作业。

依赖跟踪

理论上，Chronos 支持依赖性作业。依赖性作业仅仅在其前提条件成功运行之后才会运行。不幸的是，Chronos 依赖性作业的实现实际上有些出入。在 Chronos 里，如果父作业最近一次的调用成功了，那么就认为满足了该依赖性作业的前提条件。只要依赖性作业的所有前提条件都至少运行过一次，那么这个依赖性作业就会运行。这意味着除非每个父作业的运行间隔严格一致，否则这个依赖性作业可能就会在非预期时间运行。本书“Chronos 运维注意事项”部分会详细介绍这个问题的常规处理策略。

延迟启动

Chronos 也能够用来运行一次性的或者在未来的某个时间点重复间隔地执行作业。当指定重复执行作业的ISO 8601 间隔时，还可以指定希望什么时候首次运行该作业。这意味着用户可以使用 Chronos 在未来的每天、每周或者每月调度作业的执行，这很有助于做计划。

worker 可扩展性

Chronos 和大多数其他作业调度器的关键区别在于 Chronos 在 Mesos 上的容器内部运行其作业。这意味着作业是互相隔离的，worker 是可扩展的，因此无论 Chronos 生成什么样的负载，用户都能够添加新的 Mesos slave 来处理额外的工作，并且确保作业能够接收到所需资源。

和所有可以用于生产环境的 Mesos 框架类似，Chronos 通过热备支持高可用性——大多数 Chronos 的部署都会运行至少两个调度器的实例，来确保即使调度器或者其主机发生故障时，Chronos 也永远不会掉线超过几秒钟。Chronos 还带有一个同步工具，`chronos-sync.rb`，该工具将 Chronos 的状态同步到磁盘里，以 JSON 格式保存。使用 `chronos-sync.rb`，用户可以通过将文件提交到版本控制系统里，将要在集群里运行的作业版本化。这是确保可重复性而使用最多的方案，因为这样做允许用户在存储库里保存 Chronos 配置。

REST API 和同步到磁盘工具的组合让 Chronos 更容易集成。因为 Chronos 在一定程度上限制了特性的规模，因此很容易使用，但是它并非独立工作流管理的完整方案。

在 Marathon 上运行 Chronos

Mesos 框架部署里最常见的技术之一是在 Marathon 上运行框架调度器。这么做是因为这些调度器可以依赖 Marathon 来确保它们一直在某处运行着。这给运维人员减轻了大量负担，否则运维人员需要确保每个调度器都有良好监控，并且部署在多台机器上。

53

为了可以在 Marathon 上部署，调度器必须实现一定程度上的选主。在第 4 章里会介绍，调度器是从单台主机连接到 Mesos 上的。也就是说，如果想在两个地方运行调度器（比如，有一个热备），用户需要给出某种类型的信号确保某个时间点只有一个调度器在运行。最常见的方式是使用 ZooKeeper 来解决这个问题，因为 ZooKeeper 带有一个简单集成的选主组件（细节见“增加高可用性”部分）。Marathon 和 Chronos 都是采用的这种方案。

让我们一起看看 Marathon 上启动 Chronos 的样例 JSON 表达式（示例 3-16）。为了

最大化端口分配的效率，允许 Chronos 绑定任意端口。要连接到 Chronos 上，要么在 Marathon 的 UI 上点击，要么使用之前介绍的 HAProxy 服务发现机制之一。

示例3-16 Chronos JSON描述符

```
{
    "cmd": "java -jar chronos.jar -Xmx350m -Xms350m ---port $PORT0", ❶
    "uris": [
        "http://fetchable/uri/with/chronos.jar" ❷
    ],
    "cpus": 1, ❸
    "mem": 384, ❸
    "instances": 2, ❹
    "id": "chronos",
    "ports": [0], ❺
    "constraints": [["hostname", "UNIQUE"]] ❻
}
```

❶ 假定所有 Mesos slave 上都已经安装了 Java。因此只需要将动态分配的端口传递给 Chronos，它就能成功启动了。

❷ 与其在每台 Mesos slave 上都安装 Chronos，不如将其 .jar 文件存储到某个已知位置。这样 .jar 文件会被动态下载到被分配运行 Chronos 调度器的任意 slave 上，从而简化部署的复杂度。

❸ 选择匹配允许 JVM 分配的内存总量（为 JVM 本身预留了 34MB 的 headroom）的容器大小。

❹ 运行 Chronos 两个实例就可以实现几乎即时的故障转移，因为备用 Chronos 调度器也会一直运行着。

❺ Chronos 仅仅需要分配一个端口。

54

❻ 使用 UNIQUE 主机约束，确保备用 Chronos 实例运行在不同的机器上。如果 slave 有属性指定每个 slave 所在的机架，最好将调度器及其热备运行在不同的机架上。

验证 Chronos 在运行的最简单方式是从 Marathon 应用程序视图里点击分配的端口数量。每个任务的端口分配都是一个主机和端口的组合链接，点击会加载 Chronos UI。Chronos UI 有些已知的问题，因此推荐使用 `curl` 或者 `chronos-sync.rb` 来配置 Chronos 实例。

Chronos 运维注意事项

Chronos 是 Mesos 生态系统的重要一环，但是，要注意不能过度依赖它。在生产环境里运行 Chronos 时需要注意以下事项：

1．依赖性作业的特性并没有什么用——一定要使用工作流管理器。

2．Chronos 仍然是一个复杂的分布式系统——只有当需要在容器里可扩展地运行作业时才有必要使用。

几乎没有人使用 Chronos 依赖性作业的特性。相反，大多数人使用 Chronos 来调度流行的数据工作流工具的调用。对于简单工作流（比如顺序运行一些流程）而言，Bash 脚本是常用的方案，但是，如果超过 20 行的话，Bash 脚本就会很笨重，对于复杂循环和条件的语法支持也很弱。`make` 是另一种表达依赖关系的流行方式，因为其支持工作流的高级语法，并且支持基本的并行机制。Luigi 是功能更为丰富也更为强大的工作流管理器。它支持 HDFS 和数据库里的检查点，可以改进中间节点失败作业重试的效率，因为不需要重做检查点之前的工作。无论你的特别需求和用例是什么，都必须使用一些其他工具来管理 Chronos 负责调度的作业的执行顺序。

如果已经完成了所有上述搭建，就几乎已经可以在生产环境使用 Chronos 了！最后需要考虑的是，是否真的需要 Chronos 从 Mesos 得到的可扩展隔离性。即使已经运行着一个 Mesos 的集群，也并不一定需要使用 Chronos 来做任务管理。有很多其他作业调度服务器在 Marathon 上能够更容易地运行，仅仅在每个作业都运行在自己的容器里时才需要使用 Chronos。

55

Marathon 上的 Chronos：小结

Chronos 是一个强大的工具，帮助 Mesos 集群实现可靠性，可扩展性以及基于间隔的作业调度。它在很多公司（包括 Airbnb）里帮助强化了数据生成流程。要记住本书介绍的一些略显怪异的模式、变通方法以及建议！

Marathon 上的 Chronos 仅仅是一个例子，展示了 Marathon 如何提供高可用以及如何简化其他 Mesos 调度器的部署。一个商业化的 Mesos 产品——DCOS，在该领域更进一步，要求所有 DCOS 上的框架都由 Marathon 托管。不管部署的是 Chronos、Kafka 还是 Cotton、Marathon，都是托管其他 Mesos 框架调度器的绝佳选择。

Marathon+Chronos 的备选方案

当然，Marathon（和 Chronos）无法满足所有的集群需求。在本节中简单看看其他两种流行的 Mesos PaaS 框架：Aurora 和 Singularity。这两种框架学习曲线都更为陡峭，但是它们也提供了更多额外的特性。

Singularity

Singularity 是 HubSpot 开发的一个 Mesos 框架。它提供了像 Marathon 一样的部署服务的能力（包括 Docker 化的应用程序）。在这之上，如果某个服务在部署后无法通过其健康检查，Singularity 会自动将部署回滚。和 Chronos 类似，Singularity 也支持重复性作业，以及一次性作业，可以通过 REST API 触发。这样的设计简化了给任何服务或者重复性作业添加容器化，异步后台的处理过程。在管理服务发现和代理方面，Singularity 和另一个称为 Baragon 的工具集成，Baragon 管理并且集成 HAProxy、Nginx 以及 Amazon 的 Elastic Load Balancer。

Aurora

Aurora 是 Twitter 开发的 Mesos 框架。Aurora 带有由 Python 编写的特别先进的作业描述 API。这个 API，称为 Thermos，允许用户指定如何构建并且安装在 Mesos 执行器里运行的应用程序，以及工作流和流程序列。Aurora 有高性能的分布式服务发现 API，支持 C++、Java 和 Python 的绑定。能够运行 cron 风格语法的重复性作业。和 Singularity 类似，Aurora 还能够自动检测什么时候服务的健康检查开始失败，然后回滚到之前工作的版本。Aurora 最为强大的特性是多租户控制：Aurora 允许一些任务使用空闲的集群能力，并且随后在需要调度高优先级生产任务时将这些任务取代。还可以在用户间强制使用 quotas。这些特性都来自于在大规模 Mesos 集群上运行由上百个开发人员编写的海量应用程序的企业需求。如果你为之构建 Mesos 集群的企业听上去属于这一类型，那么便值得深入研究 Aurora。

56

本章小结

本章学习了如何使用 Mesos 构建标准、无状态、基于服务器的应用程序。并且构建出了自修复、自重启且十分简单的服务发现——在 Mesos 构建了一个基本的 PaaS！这些都是通过使用 Marathon，一种流行的 Mesos 框架而实现的。

Marathon 跨 Mesos 集群启动、监控，并且重启进程。它简化了这一切，让用户点击按钮就可以实现扩展伸缩，并且提供了很容易实现集成的 JSON API。

本章首先介绍了如何启动 Marathon，如何保护它，以及如何确保其是高可用的。随后，介绍了如何编写 Marathon 服务描述符，允许用户启动无处不在的 `SimpleHTTPServer`。这仅仅是一个示例应用程序，任何其他应用程序都可以通过简单变更命令行而启动。还介绍了如何编程查询 Marathon 上应用程序的状态。

接下来，本书深入探讨 Marathon 上应用程序的扩展。最重要的是，讨论了位置约束，以及如何轻松确保应用程序运行在正确的 slave 机器组上，并且在机架间均匀分布。

Docker 是最流行的应用程序容器化技术。本章介绍了如何在 Marathon 上花最小的代价启动 Docker 化的应用程序，以及如何配置各种 Docker 特定的参数，比如加载主机磁盘。

如果不包含健康检查、动推送和回滚，应用程序就不足够健壮。本章讨论了如何配置 TCP、HTTP 和基于命令行的监控检查，以及在升级中如何指定应用程序的行为。还探讨了从 Marathon 里如何构建自定义集成来订阅某个事件流。

然后，学习 Marathon 和服务发现代理的集成。Bamboo 是一种开源 HAProxy 配置管理器，是同步 HAProxy 和 Marathon 任务的强大工具。本章还介绍了使用 HAProxy 和 Marathon 构建可扩展、多语言微服务架构的流行方案。

57 最后，介绍如何使用 Marathon 托管其他 Mesos 框架的调度器，使用一种分布式，类似于 cron 的框架 Chronos 作为示例。还概览了 Marathon 的可替代 PaaS 框架，这些框架提供了很多额外的特性（代价是学习曲线更陡）。

可能 Marathon 及其备选方案都不能满足你的应用程序的需求，还需要对如何启动进程，或者如何处理失败等这些有更多的控制。这样的话，就需要编写自己的框架。下一章会介绍构建 Marathon 的技术，这样大家就可以构建出属于自己的自定义 Mesos 框架。

第 4 章

59

为Mesos创建新的框架

Mesos 框架是分布式系统的理念性聚合。但是这对框架开发人员而言到底意味着什么呢？如何能够真正地理解所编写的代码结构是怎么映射到 Mesos 集群里的呢？本章深入介绍 Mesos 架构，并且学习框架设计领域的一些常见模式。

本书并不试图让大家一次性能够理解 Mesos 框架的所有内容，而是会从最简单的例子开始：仅包含一个调度器的框架，没有自定义的执行器。这种类型的框架可以启动 worker 来处理进入队列的进程请求，也可以管理一个服务池。

调度器

调度器是直接和 Mesos 主 master 交互的组件。调度器有四大职责：

1. 在接受到的 offer 上启动任务。
2. 处理这些任务的状态更新，特别是响应任务故障和崩溃。
3. 持久化状态，并且管理故障恢复，从而实现高可用性。
4. 和客户端、用户，或者其他系统交互（因为没有任何系统能够孤立存在！）。

上述职责中的一些可以通过和 Mesos API 交互来实现，另外一些则必须使用所选择的平台语言来实现。比如，启动任务以及处理状态更新都是由 Mesos 回调和 API 请求来实现的。另一方面，由用户自己决定是否启动 web 服务器（比如 Java 的 Jetty，或者 Python 的 Flask）来允许客户端和调度器交互。高可用性通常要求同时使用 Mesos API 和其他 60
库函数来创建适用于问题域的解决方案。

但是如果仅仅关注调度器的话，用户怎么能启动任务呢？为了方便起见，Mesos 提供了

CommandExecutor，这是特别设计用来简化启动、编排命令行程序以及 Docker 容器的。下文会介绍几种常见的调度器抽象的高层级设计：构建服务器池，工作队列，以及作业处理器。

首先介绍服务器池的设计。其架构如图 4-1 所示。

服务器池调度器

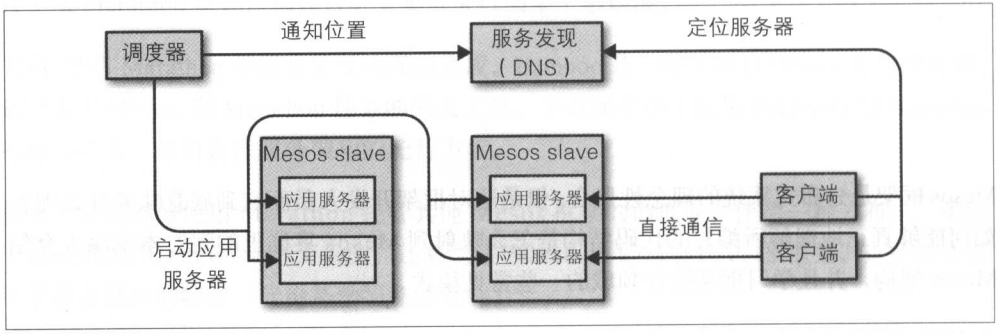


图4-1 服务器池调度器的架构和信息流

该调度器可以确保某些应用程序（比如 Ruby on Rails）一直有若干份拷贝正在运行。要构建服务器池，调度器会有想要在之上运行的所有服务器的列表。一开始，该列表包含所有处在 pending 状态的服务器。当调度器接收到 offer 时，会尝试启动服务器，将其更改为 staging 状态。一旦服务器启动成功，调度器再将其改变为 running 状态。如果服务器崩溃了，则重新置为 pending 状态，这样之后可以在新 offer 上重启。注意这里有三种状态：pending，staging，running。

可能大家会觉得 staging 状态是多余的，因为它看上去和 pending 十分类似——在这两种状态下，服务器都还没有启动。但是如果不使用 staging 状态，就会遇到一个问题：Mesos 是异步系统！在尝试启动任务和任务实际启动完成之间可能会接受到很多 offer。因此，有了 staging 状态才能确保每个服务器仅仅启动一次，并且只在启动失败后才会重试。如果没有 staging 状态，可能就会发现，某个任务本该只有一个实例，却启动了成千上万个实例。

61

工作队列调度器

如图 4-2 所示，工作队列架构允许客户端向队列提交“工作事项”，队列里的工作最终会由 worker 处理。工作队列调度器会确保一直有足够数量的 worker 运行着，从而有足够数量的 worker 来处理队列里的工作事项。

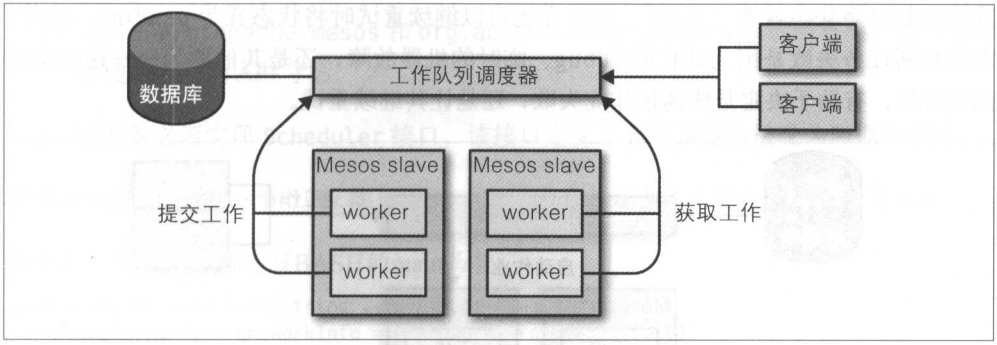


图4-2 工作队列调度器的架构和信息流

本书示例采用了一些小技巧来构建这样的调度器，假定已经拥有了上一小节所述的服务器池调度器。这样，在池里可以简单地将 worker 作为服务器启动。从而整个系统就都存在于 worker 之上了。

worker 会从已知位置获取工作事项。该位置可以作为命令行参数或者环境变量从调度器里传递给 worker。每个 worker 都会连接到队列上，然后永远进行这样的循环：获取事项，处理事项，并且将事项标记成已完成状态。

这样架构工作队列（和所有可能的框架）的好处是甚至不需要编写调度器——可以仅仅使用 Marathon 或者其他服务池的调度器。当然，也就等于将队列的实现交给了其他系统，比如 RabbitMQ 或者 Redis。这么做是有好处的：Mesos 消息 API 是设计用来控制集群上的应用程序的，不太适合其他领域，比如事务性逻辑或者高性能消息。当为 Mesos 开发框架时，用户必须使用合适的数据存储后台系统——队列、数据库和文档存储都很有用，取决于想要构建什么样的框架。

作业处理器调度器

62

最后需要考虑的抽象层是作业处理器调度器（图 4-3）。

作业处理器能从客户端接收作业列表，然后在集群里运行这些作业。一个作业就是一个待调用的命令行，及其所需的 CPU 和内存资源。该调度器需要一个略显复杂的状态机，因为用户还需要知道作业相关的终止状态（称为“成功”和“失败”）。用户还需要跟踪重试了多少次，这样可以在作业没有希望成功执行时放弃执行。

该调度器的状态机会在它尝试在某个 offer 上启动作业时，将作业从 pending 状态变更为 staging 状态，并且当作业实际开始运行时将状态从 staging 改为 running。处理任务的完成需要一些技巧，但是，用户需要在作业成功时将状态置为“成功”，在作业超过最大

重试次数时将状态置为“失败”，或者在还可以继续重试时将状态置为 pending。此外，需要判断任务失败是因为其代码的 bug、临时的机器故障，还是其他原因——这些反馈都很有用，有助于决定是快速使作业失败，还是让其继续重试。

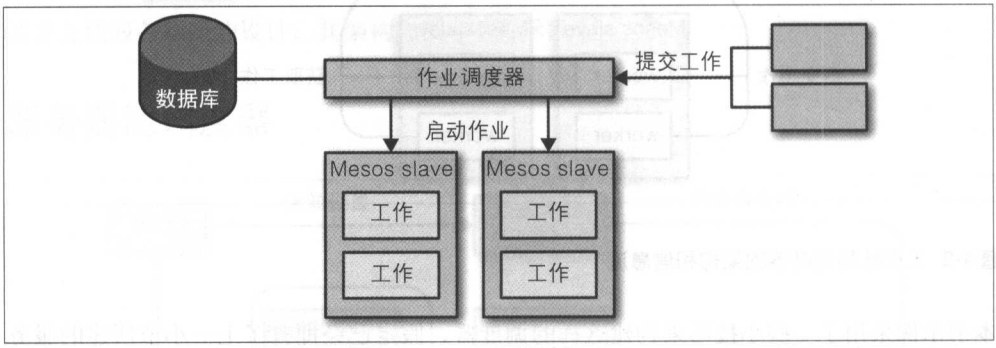


图4-3 作业处理器调度器的架构和信息流

既然上文已经证明，一个框架理论上可以仅仅使用一个调度器来实现，那么下面就一起试试，具体怎么来实现这个作业处理器。

63

没什么用的远程 BASH

在开始学习作业处理器之前，我们先看看要在 Mesos 上启动一个任务所需要的绝对最小代码是什么。我们称该程序为没什么用的远程 BASH，[1] 因为它只是在集群上的容器里运行硬编码的程序。

这里将这个最简单的框架分解成几部分，并且看看为什么每个部分都是必需的。在示例 4-1 里，可以看到这个新调度器的导入和类声明。

示例4-1 没什么用的远程BASH程序的导入和类声明

```
package com.example;
import java.nio.file.*;
import java.util.*;
import org.apache.mesos.*;
import org.apache.mesos.Protos.*;

public class UselessRemoteBASH implements Scheduler {
    // 之后会讨论这里的细节
}
```

有两个地方需要注意：

1 该框架的名称指的是它硬编码了所运行的命令，作为如何使用 Mesos API 的示例，这个程序非常有用。

1. 导入了 org.apache.mesos 和 org.apache.mesos.Protos 里的所有类，这样就可以使用 Mesos API 了。

2. 调度器必须实现 Scheduler 接口，该接口定义了调度器能够接受为回调的消息。

使用辅助的 SchedulerDriver 类给 Mesos 发送消息。示例 4-2 是应用程序的进入点。

示例4-2 没什么用的远程BASH的main函数

```
public static void main(String ... args) throws Exception {
    FrameworkInfo frameworkInfo = FrameworkInfo.newBuilder()
        .setUser("")
        .setName("Useless Remote BASH")
        .build();

    Scheduler mySched = new UselessRemoteBASH();
    SchedulerDriver driver = new MesosSchedulerDriver(
        mySched,
        frameworkInfo,
        "zk://" + args[0] + "/mesos"
    );
    driver.start();
    driver.join();
}
```


在 main 函数里，首先构造框架的 FrameworkInfo。FrameworkInfo 是向 Mesos 描述框架的 protobuf。用户需要给框架定义名称，这样才能在 Mesos UI 里找到自己的框架。实际名称里可以包含几类数据：这是什么框架，以及其他框架相关的高层级信息，比如它是 dev 还是 prod 实例。框架名称完全由用户决定，但是用户要记得考虑下述提醒！

不合适的框架名称什么时候会造成麻烦

我们曾决定在 Mesos 集群上运行 Spark，而且每个用户都能够拥有自己的 Spark 实例。第二天，其中一个用户抱怨任务无法被调度。我们打开 Mesos UI 想看看怎么回事，但令人郁闷的是，每个用户的 Spark 框架都叫做 Spark 0.7.6，完全无法辨别到底是哪个用户抢占了资源！从此以后，我们将 Spark 框架的名称修改为 $username Spark 0.7.6，这样就能快速看出集群里正在发生什么。

通常，可以将框架用户设置为一个空字符串（“”），也就是说无论哪个用户运行调度器，该框架都会注册到 Mesos 上。第 2 章的“角色”一节讨论了使用框架的用户如何来分配配额、做资源计划和保证安全性。FrameworkInfo 的大多数其他字段都是用来配置高可用性的，本书会在后面的章节再详细讨论这一块。

构造出FrameworkInfo之后，就可以创建driver了，这是实际和Mesos通信的组件。调度器driver是SchedulerDriver的实例，现在必须使用Mesos构建的MesosSchedulerDriver。将来可能会编写其他的实现，以便移除对原生代码的依赖。不幸的是，致力于该领域的现有项目给框架开发人员带来的麻烦远比价值更多，因为还很不
65 稳定。[1]SchedulerDrive有三个参数：Mesos Scheduler接口的实现，FrameworkInfo，以及Mesos集群的地址。这里，当调用start()时，driver会连接到Mesos集群上，注册框架，并且开始接受offer。通常会在main函数的最后调用join()。join()会一直阻塞直到driver终止为止，要么是接收到stop()调用，要么是因为框架故障Mesos让driver停止了。

启动SchedulerDriver的常见模式

通常，主函数的最后都会在join()之后调用start()。因为这么做太常见了，所以可以直接调用run()，它实际上就是start() && join()。

上文介绍了如何启动框架，现在一起看看示例4-3。该函数展示了如何创建能够启动的Mesos任务。因为这是一个没什么实际用途的框架，所以直接硬编码了通常作为重要参数传入的几个数值：cpu、mem，以及要运行的command。

示例4-3 没什么用的远程BASH的makeTask

```
public static TaskInfo makeTask(SlaveID targetSlave) {
    double cpus = 1.0;
    double mem = 100.0;
    String command = "echo hello world";

    UUID uuid = UUID.randomUUID();
    TaskID id = TaskID.newBuilder() ❶
        .setValue(uuid.toString())
        .build();
    return TaskInfo.newBuilder() ❷
        .setName("useless_remote_bash.task " + id.getValue())
        .setTaskId(id)
        .addResources(Resource.newBuilder() ❸
                .setName("cpus")
                .setType(Value.Type.SCALAR)
                .setScalar(Value.Scalar.newBuilder().setValue(cpus)))
        .addResources(Resource.newBuilder() ❸
                .setName("mem")
                .setType(Value.Type.SCALAR)
                .setScalar(Value.Scalar.newBuilder().setValue(mem)))
66      .setCommand(CommandInfo.newBuilder().setValue(command)) ❹
        .setSlaveId(targetSlave) ❺
```

1 致力于纯原生Mesos绑定的项目有Java的Jesos、Python的Pesos和Go的Go bindings。

```
        .build();
}
```

❶ 首先，构造一个 TaskID。TaskID 对于框架而言必须唯一确定，就像对于 slave 而言 ExecutorID，对于集群而言的 SlaveID 必须唯一一样。最为简便的做法通常是在需要生成的任意 ID 上附加上 UUID，因为这么做能够保证达到 Mesos 所要求的唯一性限制条件。

❷ 接下来，构造 TaskInfo，这是解释如何启动任务的 protobuf。TaskInfo 要求为任务设置名称。该名称是展示在 UI 上的名称，很多 Mesos 工具都会与之交互，比如 Mesos CLI。任务名称最好包含一些有用的特定信息，比如框架以及任何相关实体（比如 Marathon 上的 app 或者 Hadoop 里的 job）的名称。使用最通用（比如框架）到最特别（比如任务实例）的名称时，有些工具能够辅助提供任务名称的 Tab 键补全和自动补全。

❸ 没有任何资源的任务是没有用的：TaskInfo 记录该任务会使用的所有资源。永远需要指定的两种资源是 cpus 和 mem（注意 cpus 最后的 s）；如果应用程序会侦听网络，还需要指定 ports。还有很多 Mesos 能够管理的其他资源，第 2 章的“资源”部分有详细介绍。

❹ 因为该框架非常简单，这里设置了命令，这样 Mesos 的 CommandExecutor 会实际处理任务的运行。在 CommandInfo 里指定命令并且提供给 setcommand，本章的“CommandInfo”部分详细讨论了其特性。对于 CommandExecutor 而言，TaskID 和 ExecutorID 是相同的。

❺ 最后，将 TaskInfo 绑定到 offer 上。因为可以将同一个 slave 里的多个 offer 应用到一个任务上（构建更多总量资源的池），这里指定想要在其上启动任务的 slave 的 SlaveID，而不是想要使用的 OfferID。当实际调用 launchTasks() 时，这正是将 offer 链接到 TaskInfo 上的地方（详见示例 4-5 中组合 offer 的介绍）。

至此，框架就已经注册成功了，同时我们也学习了如何指定启动任务。最后一步是实现 Scheduler 接口。对于这个超级简单的框架，本书会忽略大多数回调函数，因为并不会使用这些回调函数所更新的信息。注意所有回调函数都会提供 driver，确保该 driver 是 start() 所调用的相同 driver。 67

Mesos 为用户传递这些，因为这是可以和集群交互的唯一方式，通常也是用户回应集群事件时想要做的事情。

示例 4-4 展示了想要记录的回调函数：registered 和 statusUpdate。

示例4-4 调度器回调日志

```
public void registered(SchedulerDriver driver, FrameworkID frameworkId,
                       MasterInfo masterInfo) {
    System.out.println("Registered with framework id " + frameworkId);
}

public void statusUpdate(SchedulerDriver driver, TaskStatus status) {
    System.out.println("Got status update "+status);
}
```

成功注册到 Mesos master 上后，就会得到 `registered` 回调函数。第一次注册框架时，无须指定 FrameworkID，Mesos 会为用户指派一个 id。上面这个回调函数是得到 `FrameworkID` 指派值的唯一途径（示例 4-15 和 4-14 里将展示如何使用这个 `FrameworkID` 让调度器安全地故障转移到新实例上，这样才能够构建出零掉线时间的系统）。`MasterInfo` 告诉用户 master 运行在哪个端口和版本上。框架能够基于 master 的版本决定哪些 Mesos 的特性是有效且可以使用的。

对于调度器而言，`statusUpdate` 回调函数非常重要，因为这是用户跟踪任务生命周期的方式：任务开始运行，以及任务失败和为什么失败。这里并不用这个回调函数处理任何事情，只是在回调被调用时将信息记录到日志里，这样做有助于框架的调试，并且帮助确认实际启动和完成的任务。构建生产环境框架时，一定要记住确保在每个 `statusUpdate` 回调里启用日志：这些日志能够帮助确定并且调试问题。

最后，示例 4-5 探讨新框架的核心。回调函数是从 Mesos 里接受 offer 的方式——这些 offer 带着诱人的资源，用户可以在其上启动任务。

示例4-5 resourceOffers的实现

```
private boolean submitted = false;

public void resourceOffers(SchedulerDriver driver, java.util.List<Offer> offers) {
    synchronized (this) {
        if (submitted) { ❶
            for (Offer o : offers) { ❷
                driver.declineOffer(o.getId());
            }
            return;
        }
        submitted = true;
        Offer offer = offers.get(0);
        TaskInfo ti = makeTask(offer.getSlaveId()); ❸
        driver.launchTasks( ❹
            Collections.singletonList(offer.getId()),
            Collections.singletonList(ti)
        );
        System.out.println("Launched offer: " + ti);
    }
}
```

❶ 在框架里，首先检查任务是否已经提交。

❷ 如果已经提交，只需简单拒绝所有 offer。

❸ 如果还没有提交，就为之构造一个 TaskInfo，加上所接受到的第一个 offer 的 slave 的 ID 信息。

❹ 随后，调用 launchTasks 启动任务！

组合 offer

launchTasks 接受 offer 列表为输入，这就允许用户将一些相同 slave 的 offer 组合起来，从而将这些 offer 的资源放到池里。它还能接受任务列表为输入，这样就能够启动适合给定 offer 的足够多的任务。注意所有任务和 offer 都必须是在同一台 slave 上的——如果不在同一台 slave 上，launchTasks 就会失败。如果想在多台 slave 上启动任务，多次调用 launchTasks 即可。

示例 4-6 里的其他 Scheduler 回调函数，现在可以先行忽略。

示例4-6 没什么用的远程BASH所忽略了的回调函数

```
public void disconnected(SchedulerDriver driver) { }
public void error(SchedulerDriver driver, java.lang.String message) { }
public void executorLost(SchedulerDriver driver, ExecutorID executorId,
                         SlaveID slaveId, int status) { }
public void frameworkMessage(SchedulerDriver driver, ExecutorID executorId,
                             SlaveID slaveId, byte[] data) { }
public void offerRescinded(SchedulerDriver driver, OfferID offerId) { }
public void reregistered(SchedulerDriver driver, MasterInfo masterInfo) { }
public void slaveLost(SchedulerDriver driver, SlaveID slaveId) { }
```

Bingo！第一个 Mesos 框架就完成了，它可以完成不少事情：能注册到 master 上，创建一个 TaskInfo 来表示如何启动任务，记录状态更新，以及接受一个 offer 来启动任务。 <69>

executorLost 没有做任何事

不幸的是，其中有一个调度器回调函数实际上没有实现，或者说 Mesos 并不支持：executorLost。如果想要在某个执行器关闭时收到通知，“Canary 任务”部分详细讨论了一种方案。这个问题在 MESOS-313 里得到了解决，因此 Mesos 的未来版本会实现这个回调函数。

这个框架实际还有两个会造成问题的 bug：当其接受 offer 去启动任务时，如果 offer 实际上太大，框架就会失败。如果 offer 太小，则会得到该任务的 TASK_LOST 状态更新，而不是得到任何表示成功的信息。另一个 bug 是当它接受了 offer 后，并不会拒绝在

resourceOffers 回调函数里接受到的其他 offer。这些其他 offer 可能就会阻塞在中间状态直到过期为止，这可能会是几分钟（或者永远，如果没有改变默认值的话）。在这段时间内，这些 offer 对于其他框架都是不可用的。

在接下来的小节里我们会改进 resourceOffers 来解决这些问题。

实现基本的作业处理器

本节会扩展示例框架增加更多功能，从 JSON 文件里读入作业列表，然后在 Mesos 上启动所有作业。首先讨论如何建模一个作业（示例 4-7）。

示例4-7 作业的实现

```
public class Job {
    private double cpus; ❶
    private double mem; ❶
    private String command; ❶
    private boolean submitted; ❷

    private Job() {
        submitted = false;
    }

    public TaskInfo makeTask(SlaveID targetSlave) { ❸
        UUID uuid = UUID.randomUUID();
        TaskID id = TaskID.newBuilder()
            .setValue(uuid.toString())
            .build();
        return TaskInfo.newBuilder()
            .setName("task " + id.getValue())
            .setTaskId(id)
            .addResources(Resource.newBuilder()
                    .setName("cpus")
                    .setType(Value.Type.SCALAR)
                    .setScalar(Value.Scalar.newBuilder().setValue(cpus)))
            .addResources(Resource.newBuilder()
                    .setName("mem")
                    .setType(Value.Type.SCALAR)
                    .setScalar(Value.Scalar.newBuilder().setValue(mem)))
            .setSlaveId(targetSlave)
            .setCommand(CommandInfo.newBuilder().setValue(command))
            .build();
    }

    public static Job fromJSON(JSONObject obj) throws JSONException { ❹
        Job job = new Job();
        job.cpus = obj.getDouble("cpus");
        job.mem = obj.getDouble("mem");
        job.command = obj.getString("command");
```

```
        return job;
    }

    // 省略无关代码 ❺
}
```

❶ 可以看到，该示例参数化了 CPU、内存以及命令参数。

❷ 将 submitted 字段移动到 Job 内，这样就可以跟踪其生命周期了。

❸ 将 makeTask 函数也移动到 Job 内，这样可以访问 Job 的所有本地字段。

❹ 为方便起见，通过工厂方法 fromJSON 定义了一种方式来构造 Job。

❺ 这里跳过了 getter 和 setter 的定义。

Job 是某个特定作业相关信息的容器，就像 MVC 里的模型类。这里仅仅添加了两个特别的方法：fromJSON，该方法能够构造带有有效起始状态的 Job；makeTask，该方法能够轻松将其转换成 TaskInfo，无须在调度器类里编写所有 protobuf 代码。

接下来，看看 main 函数是如何演进的。示例 4-8 展示了 main 函数的改进版。 71

示例4-8 没什么用的远程BASH的main函数演进版

```
public static void main(String ... args) throws Exception {
    byte[] data = Files.readAllBytes(Paths.get(args[1])); ❶
    JSONObject config = new JSONObject(new String(data, "UTF-8")); ❷
    JSONArray jobsArray = config.getJSONArray("jobs");
    List<Job> jobs = new ArrayList<>();
    for (int i = 0; i < jobsArray.length(); i++) { ❸
        jobs.add(Job.fromJSON(jobsArray.getJSONObject(i)));
    }

    System.out.println(jobs);

    // 省略无这下面的功能和之前几乎一样关代码
    FrameworkInfo frameworkInfo = FrameworkInfo.newBuilder()
        .setUser("")
        .setName("Useless Remote BASH")
        .build();

    Scheduler mySched = new UselessRemoteBASH(jobs); ❹
    SchedulerDriver driver = new MesosSchedulerDriver(
        mySched,
        frameworkInfo,
        "zk://" + args[0] + "/mesos"
    );
    driver.start();
    driver.join();
}
```

❶ 这里允许用户将文件的全部内容读入到内存里。假定第一个参数仍然是 Mesos 集群，第二个参数是包含 JSON 配置的文件的名称。

❷ 这一步里将 byte 数组转变为 JSON，这样能够抽取出 Job 配置。

❸ 最终，遍历 JSON 的 job 描述符，将其处理成 Job。

❹ 这里，向 Scheduler 传递一个参数：想要启动的作业。

72 这是作业的示例 JSON 文件，体会一下所要求的结构：

```
{
    "jobs": [ ❶
        {
            "cpus": 0.5, ❷
            "mem": 100,
            "command": "sleep 60; echo hello world"
        }
    ]
}
```

❶ JSON 文件必须包含一个带有单键、job 的 JSON 对象。该键的值必须是 Job 对象的列表。

❷ 每个作业对象有三个属性：cpus、mem 和 command，对应于 Job 类里的同名字段。

现在已经知道了如何加载作业，接下来看看如何改进 resourceOffers。其新版本见示例 4-9。

示例4-9 resourceOffers的改进版实现

```
public void resourceOffers(SchedulerDriver driver, java.util.List<Offer> offers) {
    synchronized (jobs) {
        List<Job> pendingJobs = new ArrayList<>(); ❶
        for (Job j : jobs) { ❷
            if (!j.isSubmitted()) {
                pendingJobs.add(j);
            }
        }
        for (Offer o : offers) { ❸
            if (pendingJobs.isEmpty()) { ❹
                driver.declineOffer(o.getId());
                break;
            }
            Job j = pendingJobs.remove(0); ❺
            TaskInfo ti = j.makeTask(o.getSlaveId());
            driver.launchTasks(
                Collections.singletonList(o.getId()),
                Collections.singletonList(ti)
```

```
                );
                j.setSubmitted(true); ❻
            }
        }
    }
```

❶ 因为 Jobs 包含状态信息(比如是否刚刚启动),用户需要首先计算哪些作业还没有启动。

❷ 该循环帮助用户找到还没有提交的所有作业，并将其添加进 pending Jobs。 <73>

❸ 这里尝试将作业匹配到 offer 上。

❹ 如果没有更多想要启动的作业，那么直接拒绝 offer。

❺ 如果执行到这里，表示有一个作业需要启动。从这里开始，代码和之前的版本类似。

❻ 记住需要将作业标记为已提交，否则可能会错误地多次提交该作业。

这个版本的 resourceOffers 使用了几种有用的模式。每个框架都需要持续跟踪运行中的任务，以及未来想要完成的工作。对于大多数框架而言，拥有一个能够跟踪所有进行中工作的数据结构会非常有用。该数据结构包含运行着和等待运行的任务。当然，当实际想要启动某个任务时，需要计算仍然在等待运行状态的任务的特定集合。本章的后面部分，会探讨跟踪这些信息的其他架构。

每个框架都会使用的另一个模式是遍历所有 offer，将 offer 匹配到等待运行的工作上，直到 offer 或者等待运行的工作耗尽为止。到这里你可能会想：resourceOffers 有很多需要注意的地方，该框架仍然不能高效地处理这些问题！这是对的，这也正是本书会继续探讨如何解决这个问题的原因。

将任务匹配到 Offer 上

如前文所述，将任务分配到 offer 上还有些必须解决的问题。比如，必须确保 offer 带有作业所必需的资源，必须在每个 offer 上尝试启动尽可能多的任务，还需要确定优先级，决定接下来启动哪个任务。在每个 offer 上匹配尽可能多的任务是因为如果每个 offer 仅仅匹配一个任务，调度器启动任务的速率就会很低。公式如下：

每分钟启动任务数 = slave 数量 /offer 间隔

使用默认 1 秒的 offer 间隔，如果集群里有 5 个 slave，那么每分钟仅仅能够启动最多 60 个任务。但是，如果使用正确实现的 offer 处理代码，每分钟能够启动成千上万个任务！本书示例里会实现首次适配分配算法，因为该算法要求很少的代码。然后会介绍其他算法及其需要考虑的各个方面。对 resourceOffers 做如下改动： <74>

```
... // 这里代码没有改动
driver.launchTasks(
    Collections.singletonList(o.getId()),
    doFirstFit(o, pendingJobs);
);
... // 这里代码没有改动
```

这里不是直接获取作业，而是为给定 offer 计算出所有等待运行作业中能够适配的作业。该方案会得到一个任务 List，因为这里的主要目标是能够在单个 offer 上启动多个任务。

doFirstFit 的语义

`doFirstFit` 输入为 `Offer` 和 `Job` 列表，返回可以在集群上启动的 `TaskInfos` 列表。注意 `doFirstFit` 设计为移除输入 `Job` 列表里匹配出的所有 `Job`。

显然，这里最有意思的部分是 `doFirstFit` 的实现（示例 4-10）。首次适配的原理是：基于某个 offer 里可用的空间总量，会添加适合 offer 剩余空间的任务。当决定添加某个任务时，会从剩余空间里去除该任务所使用的空间。一旦 offer 无法适应更多的任务，就会从作为参数提供的 `Job` 列表里移除所有匹配上的 `Job`，并且返回将要启动的这些 `Job` 的 `TaskInfo`。

示例4-10 First-fit offer代码

```
public List<TaskInfo> doFirstFit(Offer offer, List<Job> jobs) {
    List<TaskInfo> toLaunch = new ArrayList<>();
    List<Job> launchedJobs = new ArrayList<>();
    double offerCpus = 0; ❶
    double offerMem = 0;
    // 需要从 offer 里抽取资源
    // 这在所有语言里都有点枯燥
    for (Resource r : offer.getResourcesList()) { ❷
        if (r.getName().equals("cpus")) {
            offerCpus += r.getScalar().getValue();
        } else if (r.getName().equals("mem")) {
            offerMem += r.getScalar().getValue();
        }
    }
    // 这里，将 job 打包进 offer
    for (Job j : jobs) {
        double jobCpus = j.getCpus();
        double jobMem = j.getMem();
        if (jobCpus <= offerCpus && jobMem <= offerMem) { ❸
            offerCpus -= jobCpus;
            offerMem -= jobMem;
            toLaunch.add(j.makeTask(offer.getSlaveId()));
```

```
                j.setSubmitted(true);
                launchedJobs.add(j); ❹
            }
        }
        for (Job j : launchedJobs) {
            j.launch(); ❺
        }
        jobs.removeAll(launchedJobs);
        return toLaunch;
    }
```

❶ 用该变量持续跟踪 offer 里还剩余多少资源。

❷ 必须遍历 offer 里的所有资源，找到和所关心的资源类型相匹配的名称，随后能够提取出其值。这里不能使用 map，因为可能接收到相同名称的多种资源，比如，可能会从 slave 处接受到预留和未预留的内存。

❸ 这里我们会检查 offer 里是否还有足够的剩余资源来启动任务。如果有，会在该方法的内部计算中减掉这些资源，将 `TaskInfo` 添加到即将启动的任务列表中，并且将作业标记为已提交。

❹ 为了支持 `doFirstFit` 的语义，将即将启动的所有作业添加到一个列表里，这样就可以从所有等待运行的最初作业列表里移除该列表里的所有作业元素。

❺ `Job` 提供了方法，可以通过实例状态机跟踪其实例的进度。示例 4-11 里会详细介绍这一点。

将任务匹配到 offer 上，首次适配通常是最好的算法。你可能会想，如果在更多的工作里尝试计算出匹配该 offer 的优化组合，可能比首次适配更能高效地利用 offer。这绝对是正确的，但是要考虑如下这些方面：对于启动所有等待运行的任务来说，集群里要么有充足的资源要么没有。如果资源很多，那么首次适配肯定一直都能保证每个任务的启动。如果资源不够，怎么都无法启动所有任务。因此，编写代码选择接下来会运行哪个任务是很自然的，这样才能够保证服务的质量。只有当资源刚够用时，才需要更为精细的打包算法。不幸的是，这里的问题——通常称为背包问题（knapsack problem）——是一个众所周知的 NP 完全问题。NP 完全问题指的是需要相当长时间才能找到最优解决方案的问题，并且没有任何已知技巧能够快速解决这类问题。[1]

76

实际上，这里的情况并没有那么糟糕。虽然无法完美并且快速地解决打包问题，但是有一些技术能够给出“足够好”的结果，并且实现起来并不复杂。对于一些框架而言，所有任务的瓶颈都在单一资源上：要么是 CPU（对于受计算量限制的工作负载，比如数据

1 NP 完全问题不在本书范围之内，可以参考 wikipedia 上的文章。

分析），要么是内存（比如，类似 memcached 或者 Redis 的缓存）。因此，尝试打包资源时，可以将打包问题简化到一个维度：最受限制的维度。一旦完成这样的简化，就可以使用各种技术，比如使用完整多项式的时间贪心近似方案解决 0/1 背包问题。[1]

搭建 Offer 和 Job 之间语义差别的桥梁

现在，我们已经构建出了一个很酷的框架。它以包含想要运行的作业的 JSON 配置文件为输入，并且高效地将这些作业提交到 Mesos 集群上。但是这里有一个问题——如果作业失败了怎么办？目前还只能看到终端的一些输出表明任务失败，成功，或者丢失。这里想添加重试的支持：当作业失败时，会重新提交，期望其重试时能够成功。但是最终会认为该任务真的失败了，并且放弃执行。要达到这一目的，需要扩展 `Job` 类，这样其中不仅有一个简单的 Boolean 属性 `submitted`，还会带有一个 `JobState`：

```
public enum JobState {
    PENDING,
    STAGING,
    RUNNING,
    SUCCESSFUL,
    FAILED
}
```

这里需要添加作业可能会处于的状态的数量，用来支持重试。之前，我们可以简单启动一个未提交的作业。现在则需要跟踪某个作业所处的任务生命周期的实际阶段，仅仅在作业实际失败时重新提交该作业。图 4-4 显示有效的状态转换情况。

77

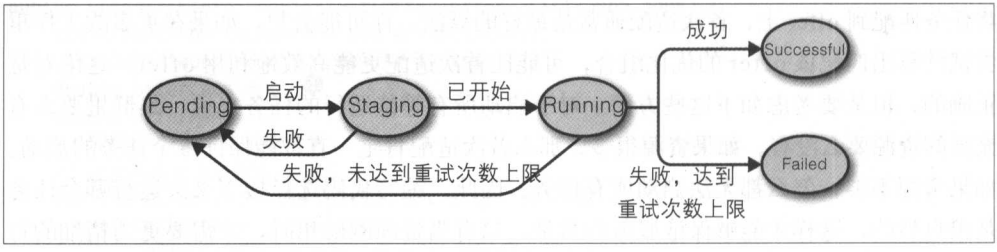


图4-4 JobState的有效转换

还需要扩展 `Job` 类，在其内部跟踪重试情况以及作业的状态。示例 4-11 展示实现了上述状态转换图的新方法。

1 这也在本书范围之外。更多信息参考 wikipedia 上关于背包问题的详细内容。

示例4-11 Job的实现

```
public class Job {
    //省略之前的字段
    private int retries;

    private Job() {
        status = JobState.PENDING;
        id = UUID.randomUUID().toString();
        retries = 3;
    }

    public void launch() {
        status = JobState.STAGING;
    }

    public void started() {
        status = JobState.RUNNING;
    }

    public void succeed() {
        status = JobState.SUCCESSFUL;
    }

    public void fail() {
        if (retries == 0) {
            status = JobState.FAILED;
        } else {
            retries--;
            status = JobState.PENDING;
        }
    }
}
```

本文示例中每个作业会重试三次。为了简化作业状态变更的实现，提供了四个方法实现状态间的迁移：launch()，started()，succeed()和fail()。向Mesos提交任务，任务开始执行，以及任务成功或失败时，会分别调用上述四个方法。示例4-12展示了如何在状态间迁移。

78

示例4-12 加强了的statusUpdate处理器

```
public void statusUpdate(SchedulerDriver driver, TaskStatus status) {
    synchronized (jobs) {
        // We'll see if we can find a job this corresponds to
        for (Job j : jobs) { ❶
            if (j.getId().equals(status.getTaskId().getValue())) { ❶
                switch (status.getState()) {
                    case TASK_RUNNING: ❷
                        j.started();
                        break;
                    case TASK_FINISHED: ❸
                        j.succeed();
                        break;
```

```
                    case TASK_FAILED: ❹
                    case TASK_KILLED: ❺
                    case TASK_LOST: ❻
                    case TASK_ERROR: ❼
                        j.fail();
                        break;
                    default:
                        break;
                }
            }
        }
    }
}
```

❶ 通常会存储 TaskID 和 Job 之间的映射关系，而不是像这里这样做线性搜索。

❷ 这里的任务状态 running，意味着已经收到了任务被 slave 成功启动的确认信息。

❸ Finished 任务指的是已经完成，没有报告任何类型错误的任务。

❹ Failed 任务指的是没有成功完成，而且是被任务本身的正常代码终止执行的任务。比如，如果应该下载某个文件的任务没能够完成，该任务会报告其状态为 failed。

79 ❺ Killed 任务指的是没有成功完成，而且是调度器让其停止的任务。当调度器执行强制占有，或者允许用户请求提前终止任务的执行时，该任务应该报告其状态为 killed。

❻ 当任务为 lost 状态时，意味着发生了预期外的错误。可能是因为 slave 意外下线，或者某个执行器在发送 TASK_FINISHED、TASK_FAILED（译者注：原书此处误为 TASK_KILLED）或者 TASK_KILLED 消息之前就退出了。

❼ 这是非常少见的情况，表明任务描述符本身有问题。详情见 TaskStatus 的 message 字段。

增加高可用性

到这里，只包含调度器的框架基本就完工了。该框架从配置文件里加载作业，在 Mesos 上启动作业，并且在作业失败时自动重试。现在是添加最后的特性——高可用性的时候了。

高可用性意味着当调度器进程失败时，框架整体仍然能够继续运行。这是通过三个子特性的合作来实现的：

1. 调度器会使用选主算法，这样可以运行多个调度器实例，并且主实例失败时备用实例能够自动接管。

2．调度器必须将其状态同步到共享的分布式存储上，这样新选举出来的主实例能够在上一个主实例失败的地方重新开始。

3．调度器必须选择一些Mesos特性，从而确保运行着的任务不会受调度器故障的影响。

本文示例使用ZooKeeper提供选主和分布式存储，因为Mesos集群通常[1]和ZooKeeper集群一起使用。示例还会进一步使用Apache Curator库和ZooKeeper交互，它能够解决所包含的ZooKeeper API的很多问题。[2]因为这并不是一个需要达到生产环境质量的框架，所以硬编码了所有框架的数据，放在ZooKeeper的`/sampleframework`路径下。对于要求生产环境质量的框架来说，所有ZooKeeper里的数据路径必须是可配置的。示例4-13展示了如何给`main`函数添加基本的选主功能。

示例4-13 简单的选主

```
public static void main(String ... args) throws Exception {
    CuratorFramework curator = CuratorFrameworkFactory.newClient( ❶
            args[0],
            new RetryOneTime(1000)
    );
    curator.start(); ❷

    LeaderLatch leaderLatch = new LeaderLatch(curator, "/sampleframework/leader"); ❸
    leaderLatch.start(); ❹
    leaderLatch.await(); ❺

    // 这里是之前介绍的主要实现
}
```


❶ 这里可以看到如何创建Curator的新实例。它接受ZooKeeper集群地址和`RetryPolicy`为参数。本书使用的是`RetryOneTime`策略，这很可能并不是生产环境上框架的正确选择。相反，`ExponentialBackoffRetry`或者`BoundedExponentialBackOffRetry`策略才是更为健壮的选择。

❷ 注意这里必须显式调用`start()`启动Curator框架。[3]

❸ 接下来，为框架创建`LeaderLatch`，会将其放到为框架选择的ZooKeeper路径`/sampleframework`下。

❹ 永远不要忘记`start()` Curator组件——这是最容易出错的#1陷阱。

❺ 启动了的`LeaderLatch`上的`await()`方法在其成为主实例之前都不会返回。

1 有项目尝试启用etcd作为ZooKeeper的替代方案。该方案由Apache Mesos Jira的MESOS-1806跟踪。
2 本书示例使用Curator 2.7.1编写，使用curator-framework和curator-recipes。
3 这里有一些混乱，Apache Curator的API也称为框架，但是，它和Mesos框架完全没有关系！

81

LeaderLatch 不错，但是 LeaderSelector 更好

`LeaderLatch` 是向应用程序添加选主的很简单的方式。只要简单调用 `await()`，然后当它返回时，它就成为主实例了！不幸的是，如果这么做，应用程序可能并不总是能够正确运行。主实例不仅仅在崩溃时才会被免职，在连不上网络时也会被免职。因此，选主更为合适的实现方式里，需要持续检查它是否还是主实例，在被免职时随后会重新创建主实例。构建健壮的选主机制的更好选择是 Curator 里的另一个类 `LeaderSelector`。`LeaderSelector` 带有一个基于侦听器的 API，在成为主实例或者被免职时会发出通知。本书没有使用 `LeaderSelector`，因为它需要更多的辅助代码。

但是，仅仅简单允许从框架的多个实例中选出主实例还不够。我们的目标是拥有新的主实例之后和之前的框架能够完全保持一致。要达到这一目的，需要让 Mesos 知道调度器支持故障转移。另外，当新的主实例注册时，需要告诉 Mesos 这是框架的新调度器。在 Mesos 里，调度器由其 `FrameworkID`、`FrameworkInfo` 里的可选值唯一确定。`FrameworkID` 必须由 Mesos 分配，从而确保对于每个框架来说该值是唯一确定的。现在，需要在分配 `FrameworkID` 时存储该值，这样未来的主实例才可以重用该值。

本书将 `FrameworkID` 存储在 `/sampleframework` 路径下的 ZooKeeper 的节点 `id` 里。启动时会首先检查是否已经存在 `FrameworkID`，如果已经存在，就会使用其启动。如果不存在，如前所述，会将 Mesos 通知的 `FrameworkID` 存储下来，这样未来的主实例就可以重用该 ID。可以无条件地存储 `FrameworkID`，因为该值在第一次分配之后就不会改变，因此它是幂等的。[1]

示例4-14 在FrameworkInfo里使用存储的FrameworkID

```
FrameworkInfo.Builder frameworkInfoBuilder = FrameworkInfo.newBuilder()
    .setUser("")
    .setName("Useless Remote BASH"); ❶
```

82

```
try {
    byte[] curatorData = curator.getData().forPath("/sampleframework/id"); ❷
    frameworkInfoBuilder.setId(new String(curatorData, "UTF-8"));
} catch (KeeperException.NoNodeException e) {
    // 不用设置 FrameworkID ❸
}

FrameworkInfo frameworkInfo = frameworkInfoBuilder
    .setFailoverTimeout(60*60*24*7) ❹
    .build();
```

❶ 和之前一样，从 builder 开始。

1 幂等的操作，无论完成一次还是一千次，效果都完全相同。当构建分布式系统时，保证尽可能多的操作是幂等的，可以在网络重新排列或者丢失消息时，降低发生竞争或者不可控行为的几率。

❷ 尝试获取存储的 FrameworkID。如果 ZooKeeper 里不存在该节点，就会抛出 KeeperException.NoNodeException。

❸ 如果不存在已经存储的 FrameworkID，就允许 Mesos 分配一个。

❹ 必须配置框架的故障转移超时时间，以秒计。本例中，允许故障转移持续一周。这意味着关闭之前的调度器之后，有一周的时间为该框架重新连接新的调度器。设置该超时时间以便系统严重受损时能有足够的时间恢复系统。超时时间达到后，使用这个旧 FrameworkID 注册的请求就会出错，提示注册的是已完成的框架，并且所有运行在该框架下的任务都会被“杀死”。

现在，需要更新 registered 调度器回调函数来存储 FrameworkID。新版本如示例 4-15 所示。

示例4-15 存储FrameworkID

```
public void registered(SchedulerDriver driver, FrameworkID frameworkId,
                       MasterInfo masterInfo) {
    System.out.println("Registered with framework id " + frameworkId);
    try {
        curator.create().creatingParentsIfNeeded().forPath( ❶
            "/sampleframework/id",
            frameworkId.getBytes()
        );
    } catch (KeeperException.NodeExistsException e) {
        //什么也不做 ❷
    }
}
```

❶ 尝试将 FrameworkID 的数据存储到之前确定的位置上。

❷ 如果捕捉到该异常，说明已经创建了节点 /sampleframework/id。这意味着之前的主调度器已经存储了这个 ID，因此不需要做任何事。如果使用的是 ZooKeeper 之外的存储，每次都简单地无条件存储 ID 可能会更为方便，因为该值永远也不会改变。 83

现在，最终得到了能够故障转移的高可用框架。至少已经能够同步哪些作业正在运行这样的状态。要达到这个目的，还需要添加支持从 ZooKeeper 序列化和反序列化作业。将每个作业存储在 ZooKeeper 的 /sampleframework/jobs/$jobid 处，这里 $jobid 是作业的 id 字段。可以通过 JSON 轻松实现序列化。

既然框架是高可用的，用户就不希望总是需要提供其必须运行的作业集合，而是希望框架能够自动监控所有之前在命令行里提供的作业，并且能够有选择地启动命令行提供的新作业。要达到这个目的，就需要能够从命令行（如果提供的话）以及从 ZooKeeper 加载作业，如示例 4-16 所示。

示例4-16 加载作业数据

```
List<Job> jobs = new ArrayList<>(); ❶

if (args.length > 1) { ❷
    byte[] data = Files.readAllBytes(Paths.get(args[1]));
    JSONObject config = new JSONObject(new String(data, "UTF-8"));
    JSONArray jobsArray = config.getJSONArray("jobs");
    for (int i = 0; i < jobsArray.length(); i++) {
        jobs.add(Job.fromJSON(jobsArray.getJSONObject(i), curator));
    }
    System.out.println("Loaded jobs from file");
}

//从 ZK 加载 job
try {
    for (String id : curator.getChildren().forPath("/sampleframework/jobs")) { ❸
        byte[] data = curator.getData()
                             .forPath("/sampleframework/jobs/" + id); ❹
        JSONObject jobJSON = new JSONObject(new String(data, "UTF-8"));
        Job job = Job.fromJSON(jobJSON, curator); ❺
        jobs.add(job);
    }
    System.out.println("Loaded jobs from ZK");
} catch (Exception e) {
    // 实例代码达不到生产环境的要求                    ❻
}
```

84

❶ 首先创建存储所有 Jobs 的 List。

❷ 只在命令行传入文件名时运行初始作业加载的代码。

❸ Curator 里的 getChildren() 方法允许用户得到某个路径下所有子节点的名称。在大多数导致状态改变的方法，比如 launch()、started() 和 succeed() 之后会保存这些子节点。

❹ 对于每个作业，从 ZooKeeper 获取序列化的 JSON 表述。

❺ 这里可以使用最初的 JSON 处理代码来反序列化 Job。注意必须给每个 Job 提供 Curator 的引用，因为 Job 必须序列化自身。

❻ 在实际生产环境代码里，还需要处理异常。

还需要加强 Job，使其能够保存状态，如示例 4-17 所示。

示例4-17 Job序列化

```
public class Job {
    省略之前定义的字段
```

```
    private String id; ❶

    private void saveState() { ❷
        JSONObject obj = new JSONObject();
        obj.put("id", id); ❸
        obj.put("status", (status == JobState.STAGING ?
            JobState.RUNNING : status).toString()); ❹
        // 简洁起见省略其他字段
        byte[] data = obj.toString().getBytes("UTF-8");
        try { ❺
            curator.setData().forPath("/sampleframework/jobs/" + id, data);
        } catch (KeeperException.NoNodeException e) {
            curator.create()
                   .creatingParentsIfNeeded()
                   .forPath("/sampleframework/jobs/" + id, data);
        }
    }

    public void launch() {
        // 省略之前的代码
        saveState(); ❻
    }

    // started(), succeed(), fail() 如上做相应修改
}
```


❶ 为每个 job 添加一个标识符，这样可以通过数据库里保持同步的主键查询。

❷ 添加方法将 Job 的状态保存到外部存储里。示例将 JSON 格式的数据保存到 ZooKeeper 里。

❸ 确保将所有字段都保存到数据库里。

❹ 将 STAGING 的作业保存为运行中作业。因为无法确认处在 STAGING 状态的作业是否已经成功启动，因此这里假定其在故障转移中成功了，这样要么在其完成时得到新的 statusUpdate，要么在核对之后得到 TASK_LOST。

❺ Curator 的 API 要求不同的方法，来创建新的 ZooKeeper 节点，并且更新其数据。

❻ 确保每次状态更新之后保存 Job 的状态。注意 started()、succeed() 和 fail() 方法里都忽略了最后需要添加的 saveState()，但是在真实框架的代码里必须记得添加。

好了！作业调度器现在就能够自动故障转移到等待着的热备实例上了，并且在绝大多数情况下，不会影响正在运行的作业，还能持续跟踪作业的信息。

添加核对

有些时候还是会有不同步的情况发生，详情见“故障转移期间的核对”部分。这是本书示例中，高可用的 Mesos 调度器最后一处还不够完善的地方。幸运的是，向调度器添加核对很容易。当实现 `statusUpdate()` 回调时，会将 Mesos 状态事件映射到框架的状态机模型上。只要这一状态机永远不从最终状态（比如 `TASK_LOST`、`TASK_FINISHED`、`TASK_KILLED` 等）移出，核对就不会导致额外的工作。通常来说，系统需要周期性地运行核对，会在启动时运行，并且在启动后每隔 30 分钟运行一次。核对是所有丢失消息的捕捉器——它确保即使一些已知 bug 或者硬件故障导致框架和 Mesos 集群的状态不同步时，系统也能够最终修复这样的不同步问题。

要实现核对，需要生成正在运行的任务列表，以及任务运行的 slave 的 ID（这么做的原因详见“故障转移期间的核对”部分）。然后，调用 `reconsileTasks()`。Mesos 会自动发送响应到 `statusUpdate()` 回调函数。如果想要实现核对相关状态更新的特别处理，可以检查状态更新的 `reason` 字段是否是 `TASK_RECONCILIATION`。在示例框架里没有考虑自定义的处理方式，因为示例框架的任务状态机很健壮。示例 4-18 展示了如何核对正在运行的任务。

示例4-18 正在运行任务的核对

```
List<TaskStatus> runningTasks = new ArrayList<>();
for (Job job : jobs) {
    if (job.getStatus() == JobState.RUNNING) {
        TaskID id = TaskID.newBuilder().setValue(job.getId()).build();
        SlaveID slaveId = SlaveID.newBuilder().setValue(job.getSlaveId()).build(); ❶
        System.out.println("Reconciling the task " + job.getId());
        runningTasks.add(TaskStatus.newBuilder()
                .setSlaveId(slaveId)
                .setTaskId(id)
                .setState(TaskState.TASK_RUNNING)
                .build());
    }
}
driver.reconcileTasks(runningTasks);
```

❶ 虽然这里没有展示代码，但是必须记录每个任务运行所在的 slave 的 ID，这样才能执行核对操作。

至此，就已经完整实现了一个高可用的 Mesos 调度器，它足够健壮，能够抵御所有故障类型：master 可以崩溃，slave 可以断网，并且调度器本身可以被“杀死”和重启。无论发生上述什么问题，框架都能够容错，能够持续跟踪、监控，并且确保其任务能够成功执行。通过如下几点可实现这样的设计：

Job 在任务之上

本书将想要从 Mesos 任务里运行的作业的概念拆分出来。这使得用户可以管理作业的重试，因为作业的成功或者失败并不仅仅依赖于单次运行。本书的概念性框架也适用于长期运行的服务：服务是一个 goal，用来运行某个程序的指定数量的实例；任务是一个实例，但是它可以失败，并且被其他任务所取代。如果对使用 goal 处理可扩展、分布式系统的可靠性的理念感兴趣，要知道该理念是由 Joe Armstrong 在其构建可靠电话系统的系列文章[1]中首次提出的。

持久化框架标识

通过保存框架的标识 `FrameworkID`，实现了用不同的调度器来自动完成故障转移。这是必须在 Mesos 之外自己实现的框架管理的一部分，因为它必须在故障发生，哪怕是 Mesos 本身的故障发生时还能够保证系统的运行。

共享作业状态

当收到某个作业状态更新时，会将这些更新同步到所选择的分布式存储上，示例里是 ZooKeeper。这确保新选举出的主调度器能够得到集群状态的最为确切的信息。

作业的重试

有时，Mesos slave 机器不工作了：可能它的配置有问题，也可能在这台 slave 上运行的其他任务，通过 Mesos 无法隔离的资源类型影响了这个任务。要防止这样的情况发生，就要能够进行操作的重试。但是，当作业持续失败时，通常需要选择放弃，因为这可能表明作业本身的确有问题。

核对

最终，事情还是会以无法预测的方式失败。为了处理这样的事件（或者至少能够自动从这样的故障中恢复），本书示例使用了 Mesos 核对 API 来周期性地尝试清理状态。虽然核对无法处理所有问题，但是它仍然是处理未知问题时的最优工具。

高级调度器技术

虽然本书已经实现了一个功能完整并且可靠的调度器，但是还可以添加更多的功能来保证调度器的健壮性。本节详细讨论要达到生产环境质量的调度器所需要考虑的改进。

1 Joe Armstrong，*Making reliable distributed systems in the presence of software errors* 一书的作者，也是 Erlang 的创造者。即使大部分程序员没有使用过 Erlang，但是其关于面向 goal 的编程和监管的思想广泛应用在可靠分布式系统中，包括 Mesos 及其竞争对手，Google 的 Borg。

88

分布式通信

本书中没什么用的远程 BASH 示例程序里，只有当调度器成为主实例，传入包含 JSON 作业描述符的文件之后才会处理新作业。真正的调度器需要在任何时间都能够接受新作业。大多数情况下，会通过 HTTP API 和调度器通信工作。

如果允许调度器接收 HTTP 的工作，那么主和热备调度器最好都能够处理 HTTP 请求。可以通过以下方式实现：

共享数据库

> 这里，主和热备调度器都能够读取并写入共享数据库。这样，它们就能够同时处理客户端的请求。但是，无论什么时候客户端修改数据库，都必须确保会通知主调度器。一些数据库支持发送通知，如果所用数据库不支持发送通知，主调度器可以频繁查询数据库更新，这样它就不会落后于客户端指令太多。解决通知问题的另一种方案是使用选主子系统来确定当前的主调度器，然后向其发送远程过程调用（RPC），这样它可以自己完成更新。

重定向修改请求

> HTTP 已经支持重定向。在这种情况下，主和热备调度器都能够处理只读请求，但是更新框架状态的请求可以简单通过 302 Found 请求重定向到主 master 上。

数据是真理

> 该模式类似于共享数据库模式，但是没有提供 HTTP API，客户端直接和数据层通信，调度器完全服从数据层里的数据。比如，如果使用 Redis 或者 RabbitMQ 作为数据层，任意客户端都能够向其中添加工作，主调度器会周期查询需要启动的新任务，并且将所有状态的更新复制到数据层。

实际上，大多数框架作者首选的是 ZooKeeper。ZooKeeper 是很多种类型框架的最佳选择，原因如下：

- 它提供选主元素。
- 它存储任意数据。
- 它支持任意事务。
- 它是高可用的。
- 当数据改动时可以通知客户端。
- Mesos 集群已经有这样一个实例。

89

ZooKeeper 的主要不足点是它无法扩展到成千上万的数据元素。这时，用户需要放弃

ZooKeeper 所提供的高度一致性，而选择其他数据存储，比如 Cassandra、Redis 或者 MySQL。

强制故障转移

有时需要让框架自动强制故障转移到新调度器上。这个特性在两种场景中很有用：

1. 调度器知道当前运行的版本，如果有更新版本就触发一次故障转移，从而简化升级流程。

2. 调度器在用户的命令行里开始运行，但是它实际上在 Mesos 集群上创建了一个会取代命令行工具的任务。比如，`mesos-submit`（现在已经不建议使用了）就这么做的。

如果想要触发一次故障转移，只要简单地使用已有 `FrameworkID` 注册新的调度器就可以了。当 Mesos 接受到新调度器的注册请求时，会触发旧调度器的 `error()` 回调函数，发送消息 `Framework failover`。不幸的是，该消息字符串是唯一能区分计划内的故障转移和实际框架错误的方式。

合并 Offer

一些调度器会遇到的一个问题是 offer 碎片化。当多个框架连接到 Mesos 集群上，并且其中一种框架启动多个类似(但是不完全一样)大小的任务时，会导致 offer 碎片化。通常，Mesos master 能够合并每台主机的空闲资源。当 master 无法完成这样的合并时就会导致碎片化。根本原因在于资源的不同部分被重复 offer 给不同的连接着的框架，因此永远都没有足够空闲的时间来完成合并。比如，一台主机有八个 offer，每个 offer 有一个 CPU，而不是一个八个 CPU 的 offer。要解决这个问题，框架必须改变处理 offer 的方式：不是立即使用或者拒绝 offer，而是能够临时等待并且查看这些主机上是否会有更多的 offer 提供出来。记住，当调用 `launchTasks` 时，可以传入同一个 slave 的任意多的 offer（参见示例 4-5 中的合并 offer）。

可以将 offer 合并构建成 `resourceOffers` 回调的中间件格式。不是直接调用 `resourceOffers` 的最初实现，而是首先将所有新 offer 放到 map 里，slave 作为键，这样可以跟踪每个 slave 的所有 offer。然后决定什么时候从 map 里移出一组 offer，并且将其传递给最初的 `resourceOffers` 回调函数。这么实现时，还必须要注意确保不会意外地永远占有某个 offer——必须在几分钟后尝试拒绝 offer，从而保证整个集群的平稳运转。

决定什么时候将 offer 组传递给最初 `resourceOffers` 实现，可以采用的策略有两大类型：在足够大，或者足够长的时候传入 offer 组。如果选择足够大的策略，需要跟踪待启动的最大任务。然后，当最终接受到某台 slave 上足够多的满足该任务的资源时，就可以将

offer 组传递给代码完成任务的启动。如果选择足够长的策略，还需要确定占有某个 offer 多长时间（通常为 15 ~ 60 秒）。然后，当某个 offer 占有时间比所选时间长时，可以将 slave 的 offer 传递给最初的 `resourceOffers`，并且期望该组足够大。

足够大策略的优势在于，当找到足够的 offer 时，一定能够启动任务。缺点是仍然需要担心活跃度（偶尔也会拒绝旧 offer），并且需要跟踪等待运行任务的规模。如果尝试将多个任务打包到一个 offer 组里，该策略就会变得更加复杂，因为需要持续跟踪很多变量：每个 slave 上的 offer、待运行任务，以及这些任务可能会被打包进 offer 的不同方式。

另一方面，足够长的策略更容易实现：用户完全无须担心任务的规模；相反，可以假定在所选的时间窗内最终会接受到足够的 offer 来满足任何待运行任务。该策略的缺点在于，启动很大的任务时会很痛苦，因为只有等待更长的时间，才有可能接受到满足条件的 offer。

最后需要强调的是，直到达到 Mesos 集群的某个极限时，才有必要合并 offer。但是，了解如何诊断问题，以及如何设计解决问题的方案是很有必要的。笔者在曾经负责运维的 Mesos 生产集群里，遇到过这样的问题，并且通过使用足够长策略成功解决了该问题。

调度库函数 Fenzo

Fenzo 是 Netflix 在 2015 年夏天发布的库函数。Fenzo 为基于 Java 的调度器提供了完整的解决方案，完成 offer 缓冲，多任务启动，以及软和硬约束条件的匹配。就算不是所有的，也是很多调度器都能够受益于使用 Fenzo 来完成计算任务分配，而不用自己编写 offer 缓冲、打包和放置路由等。

91

加固调度器

在示例调度器里，作业失败之后只是简单地在下一个 offer 上重试该作业。通常，由于某台特定机器的特别问题，一些作业可能无法在这台机器上运行。为了避免发生这种类型的重复性失败，需要跟踪哪些主机上的作业失败了，从而不再在这些主机上重试该作业。实际环境里，这么做可以避免遇到所有类型的主机错误配置的问题。

我们会遇到的主机相关的问题并不仅限于此。极少数情况下（但是规模够大时极少数也意味着很多），可能会遇到黑洞主机：会导致其上运行的每个作业都失败，通常还是很快就失败的主机。黑洞主机会迅速耗尽正在尝试运行的所有作业的重试次数，导致需要人工干预的大面积故障。要处理黑洞主机的问题，需要跟踪过去几分钟内每台主机上发生的失败次数。如果该数目超过预设定的目标值，那么就应该停止接受来自于该主机的 offer。一些框架期望任务永远也不会失败（比如长期运行的服务），因此在一分钟内有来自单台主机的 20 次失败就会将该主机标识为黑洞主机。另一方面，一些框架每秒完

成上千次任务，因此它们只会将短时间内发生上千次失败的主机放到黑名单里。最终，实现这个黑名单时，要确保当主机进入黑名单时会发出通知，并且确保一周后这些主机会自动从黑名单里释放出来。如果不自动通知，并且随后解除对这些主机的阻塞，管理员就很可能忘记这些主机，使其永远闲置，导致花费了金钱但是没有完成任何工作。也可以选择使用自动系统，比如 Satellite 来为所有框架处理该领域的问题。

Framework UI

通常，框架都带有 web UI，因为浏览器是图形界面的强大平台。`FrameworkInfo` protobuf 里带有一个称为 `webui_url` 的字段，其值应该设置为 UI 可用的 URL 地址。当设置该字段时，Mesos UI 会将 UI 里的框架名称改为指向所提供的 `webui_url` 的链接。如果使用该特性，用户必须确保要么在负载均衡器后管理该 `webui_url`，要么在新调度器成为主实例时在 `FrameworkInfo` 里更新该字段。

分配端口

上文示例只在调度器里使用了标量资源(比如CPU和内存)。Mesos还支持范围类型资源，最常见的就是端口。Mesos 管理集群上的端口，确保需要侦听套接字的每个任务都绑定到唯一的端口上，并且确保要求特定端口的任务不会在这个端口已经被占用的主机上启动。可用端口的格式是可用端口的 `[begin,end]` 列表，要使用端口，可以将感兴趣的子范围添加为资源。比如，示例 4-19 展示的功能 `findNPorts` 会从给定 offer 里找到 N 端口，并且返回必须添加到 `TaskInfo` 里的需要这些端口的资源。

92

示例4-19 端口分配

```
public static Resource findNPorts(Offer offer, int n) {
    int numStillNeeded = n;
    Value.Ranges.Builder ranges = Value.Ranges.newBuilder();
    List<Value.Range> availablePorts = null;

    for (Resource r : offer.getResourcesList()) {
        if (r.getName().equals("ports")) {
            availablePorts = r.getRanges().getRangeList(); ❶
        }
    }

    while (numStillNeeded > 0 &&
                availablePorts != null &&
                !availablePorts.isEmpty()) { ❷
        Value.Range portRange = availablePorts.remove(0); ❸
        long numAvail = portRange.getEnd() - portRange.getBegin() + 1; ❹
        long numWillUse = numAvail >= numStillNeeded ?
                numStillNeeded : numAvail; ❺
        ranges.addRange(Value.Range.newBuilder()
                .setBegin(portRange.getBegin())
```

```
                .setEnd(portRange.getBegin() + numWillUse - 1) ❻
                .build());
        numStillNeeded -= numWillUse;
    }

    if (numStillNeeded > 0) { ❼
        throw new RuntimeException("Couldn't satisfy " + n + " ports");
    }

    return Resource.newBuilder()
        .setName("ports")
        .setType(Value.Type.RANGES)
        .setRanges(ranges.build())
        .build();
}
```

❶ 首先找到包含可用端口的 Range。

❷ 只要仍然有可用端口并且需要这些端口，就会持续查找。

❸ 一次检查一个连续的子范围。

93 ❹ 连续子范围包含边界，因此，如果子范围是 [10-20]，实际上可用端口为 11 个。

❺ 当决定该子范围内使用多少端口时，永远不会超过仍然需要的数量或者可用数量。

❻ 记住范围包含边界，并且要注意 -1 是否会出错！

❼ 可能无法找到足够端口，最好抛出异常而不是返回错误数据。

除了范围之外，Mesos 还提供了集合类型的资源。资源集合目前只有在 slave 命令行上有定义的时候才存在。它们作为 Resource protobuf 里的列表存在，并且使用方式很直接。“配置自定义资源”部分详细介绍了如何创建自定义的资源类型。

检查点

虽然我们应该保证框架的实现足够健壮，能够抵御所有错误，但是实际上让 Mesos 创建检查点，将状态更新从任务里保存到磁盘上也很有必要，这么做有助于保证 slave 重启后状态的一致性。要启用该功能，需要将 `FrameworkInfo` 里的 `checkpoint` 属性设置为 `true`。这会将一些 slave 的崩溃和重启归类为普通行为。我推荐一直这么做，除非短期间隔内会生成大量状态更新。因为检查点会将每次状态更新都写入磁盘，如果每个 slave 每秒钟发生上千次状态更新的话，可能会导致每个 slave 每秒发生上千次额外的磁盘 I/O 操作，这很可能会降低集群的性能。用户如果关心性能影响的话，应该在启用和不启用检查点时对框架做基准测试；否则，最好一直使用它。

CommandInfo

CommandInfo protobuf描述进程如何被启动。本书介绍CommandInfo的两个方面：和启动进程相关的字段，以及和为进程搭建环境相关的字段。

启动进程

CommandInfo支持启动进程的两种方式：通过execv syscall，以及通过系统默认shell（比如Dash或者Bash）。使用execv启动进程能够确保用户可以精确控制传递给进程的参数，并且无须担心shell如何解析字符串。想要直接启动进程，需要将value设置为可执行文件的路径，将arguments设置为参数数组，并且将shell设置为false。不过，如果想通过shell启动（这样就可以使用shell操作符，比如&&、||和|来组织程序），则必须将shell设置为true，然后value里的整个内容可以通过运行/bin/sh -c "$value"来调用，shell命令会忽略arguments。

94

配置进程环境

Mesos允许用户控制进程环境的额外三个方面：以其身份运行的用户账户（通过user字段，受限于ACL），进程能够访问的环境变量（通过environment字段，是封装进protobuf的键/值对列表），以及文件系统状态（通过URI）。需要注意最后这个参数：它指定需要下载到执行器沙箱（执行器开始运行时的当前工作目录）里的任意数量的URI。用户可以选择这些URI是否应该是可执行的，这会很有帮助，因为用户可以指令CommandInfo启动并且将应用程序的二进制文件直接安装到沙箱里，避免使用部署系统。用户还可以指定是否解压归档文件，因为Mesos能够理解这些扩展形式，并且能够自动下载以及解包应用程序或者额外的资源。目前可以识别的扩展形式为.tgz、tar.gz、.tbz2、.tar.bz2、.txz、.tar.xz，和.zip。默认情况下，Mesos支持URI架构hdfs://、http://、https://、ftp://、ftps://、和file://（为NFS或者其他网络化的POSIX文件系统所用），以及所安装的Hadoop客户端支持的任意其他URI架构（但是，Hadoop并不是必需的依赖，如果没有安装Hadoop的话，hdfs://会失败）。用户可以通过修改src/launcher/fetcher.cpp添加获取二进制文件的额外方式。

环境变量$MESOS_DIRECTORY指定了任务的沙箱。

本章小结

本章介绍了如何实现 Mesos 框架的调度器组件。调度器负责启动任务，监控任务，并且和用户交互。很多框架仅仅需要实现调度器，通过编排 worker 进程的启动，就可以实现大多数分布式、可扩展、集群化应用程序所必需的功能。本章详细介绍了一些示例框架架构，比如服务器池调度器、工作队列调度器和作业处理调度器。随后，详细介绍了作业处理器的实现。

95 要构建作业处理器调度器，需要使用数据库 ZooKeeper 使应用程序的状态持久化。虽然调度器 API 有很多回调函数，但是用户仅仅需要实现其中一部分就可以创建出能工作的系统。我们从任务（Mesos 在集群上所运行的代码）里分离出作业（用户想要干的事情）的概念。这样的隔离使得用户能够在调度器里轻松实现重试。调度器还需要一些基本的优化——offer 打包，来避免性能过差。

调度器启动并且开始运行之后，通过组合外部选主库函数，Curator 和 Mesos 框架注册 API 的额外特性，给系统添加了高可用性。不幸的是，一旦系统能够实现故障转移，那么就有可能丢失任务状态的更新。要解决这个问题，示例实现了任务核对，这样当计划中和计划外的故障转移发生时，能够确保调度器工作正常。

介绍了如何实现可靠的 Mesos 调度器之后，本章讨论了满足生产环境要求的框架所必须考虑的额外方面。因为通常会运行着调度器的很多实例，并且其中只有一个是 master，我们介绍了一些设计，讲解客户端如何和调度器交互。还介绍了 offer 碎片化，包括是什么导致了碎片化，以及哪些策略能够确保框架永远能够找到足够大的 offer。除了 CPU 和内存，很多框架必须给其任务分配端口，这里还介绍了范围类型的资源，以及端口分配和 CPU、内存这样的标量资源的分配有什么不同。最后，简要讨论了一些其他问题，比如为框架构建用户接口，加固框架来抵御未知问题的影响，以及在维护期间自动化地实现故障转移。

至此，我们已经可以编写出一个 Mesos 框架，该框架能够启动和监控任务，处理各种类型的失败和预计外的场景，并且能够在集群上高效运行。下一章会介绍如何通过实现自定义的执行器来增加框架的功能，从而能够动态改变容器的大小，并且允许在任务间共享资源。

第5章

构建Mesos执行器

前面已经学习了如何为 Mesos 构建调度器。但是，有些事情单独使用调度器 API 不一定能够实现。比如，在相同容器内运行多个任务，可能应用程序有调度器的生命周期消息，报告其进程或者应用程序特定的统计信息，也可能想将额外功能放置到执行任务里。为了完成这些事情，需要编写自定义的 Mesos 执行器。

这正是本章要学习的内容。本章一开始会简单介绍内建 CommandExecutor 所提供的功能，最后会对此进一步深入探讨。然后添加心跳支持，帮助更快地侦测故障。最后探讨可行的设计方案，涉及进程报告和加强的日志，以及在相同容器内运行多个任务。

执行器

之前已经介绍了什么是调度器：它是和 Mesos master 以及框架客户端交互的组件，能够管理运行着的任务，并且处理故障转移。但是什么是执行器呢？执行器有三大职责：

- 执行调度器所请求的任务。
- 保证通知调度器这些任务的状态。
- 处理来自调度器的其他请求。

你应该不想这么做

编写执行器很枯燥。不幸的是，测试调度器和执行器之间通信是否正常的唯一方式是在 Mesos 集群上启动一个框架的新实例。此外，正确实现健康检查、进程管理和同步是很困难的。我强烈建议大家思考如何能够利用现有的 CommandExecutor，而不是构建自己的执行器。但是，也有很多问题只能通过编写执行器才能解决。

构建工作队列 worker

工作队列调度器需要单个 worker 执行工作事项。之前讨论这一点时，我们建议使用外部队列系统，比如 Redis 或者 RabbitMQ，worker 进程可以直接连接到上面，但是这并不是构建该系统的唯一方式。使用自定义执行器，可以直接向执行器发送任务。如果使用 RabbitMQ 或者 Redis，这么做并没有太多优势，但是，如果选择将状态存储到 Postgres 这样的数据库里，可能就无法轻松地从每个 worker 扩展到上千个连接。因此，可以简单地让调度器向相同的执行器发送多个任务。

这么做既有优势也有劣势。一方面，用户可以不用添加另一种基础框架（队列层）来解决扩展问题。另一方面，需要自己负责实现调度器里的排队语义，这比依赖那些久经考验的队列产品要难得多。

运行 pickled 任务

什么是 pickled 任务？

表示一个任务的序列化代码或者配置数据就称为 pickled 任务。比如，发送 RPC 要求序列化任务参数的方法，这样它才能跨网络传输。很多语言支持 bytecode、可执行源码或者其他数据的发送。“pickle”这个词来自于 Python 的序列化库函数。

很多用户使用 Java 虚拟机（JVM）或者 Docker 运行代码。因为可能需要较长的初始化时间，
99 所以用户想向执行器提供生成好的代码。比如，需要 45 秒才能启动带有所需类的 JVM，这样的话每秒就无法完成上千次的执行。为了避免每个任务都花费 45 秒的额外消耗，则很有必要在相同主机上的任务间重用 JVM。使用自定义的执行器，就能够从每个任务里读取元数据，并将其反序列化为可执行代码。然后，执行器就能够简单运行这些代码，重用整个 JVM。对于短时间的任务而言，这可能就是任务需要消耗 10 秒还是 55 秒的区别，这也是保证 Spark 低延时的技巧所在。

该方案还能够用来部分降低编写执行器的风险：用户仅仅需要编写一次“pickle 执行器”。完成之后，就能够受益于执行器的动态大小调整（详见“多个任务”部分），并且无须再次调试之间的通信。注意这并不要求一定要构建自定义的执行器——还可以通过带外机制和 pickled 任务通信，唯一的缺点是会丢失动态调整容器大小的功能。

共享资源

假定要编写一个应用程序，其中每个任务都会读取大量静态数据集和小部分用户提供的

数据集，处理数据，并且返回结果。这个应用程序可能用于运行一系列模拟，或者训练机器学习模型。可以使用自定义执行器轻松共享大型静态数据集，同时当执行器终止时，还能够受益于 Mesos 的资源计量和自动清理的能力。要实现这一目标，可以配置自定义执行器，使其下载大型数据集到沙箱里，并且预留足够的内存，从而保证这些数据存在于文件系统的内存内块缓存里，来完成自身的初始化。随后，在这个执行器上启动的任务就能够映射到该数据集，从而可以得到内存内直接的访问。该方案能够避免该数据集的多次拷贝，从而降低网络带宽和磁盘的消耗。

使用自定义执行器共享资源的另一个场景是构建通用的图像处理单元（GPGPU）应用程序。通常，每个 slave 上只有一个 GPGPU。如果使用 `CommandExecutor`，并设置 GPGPU 为自定义资源（详见“配置自定义资源”部分）时，那么每个 slave 上一次只能运行一个任务。但是，用户可能希望以应用程序特定的方式管理 GPGPU 的共享。要达到这一目标，可以使用上节介绍的 pickled 任务模式，确保每个任务在其需要使用 GPGPU 时能够编程式暴露出来。那么自定义执行器就能够在相同容器里运行所有任务，从而中转并调度任务对 GPGPU 的使用。

这里只是可能需要在任务间共享资源的一些例子。当然，还有很多种任务共享工作的方式，从通用的子问题到缓存的共享。有时，利用这些结构的最高效方式就是在相同容器内运行它们，这就要求编写执行器。

更好地看护

Mesos 是设计用来最大化可用性的：当调度器崩溃时，它能够自动故障转移；当 slave 升级时，执行器能够自动重连接到新版本；当网络连接断开时，slave 和执行器能够继续运行最后已知的指令。但是，对于一些应用程序而言，这些还不是它们实际想要的。

大规模部署后，部分预计中以及预计外的故障会暴露测试和 QA 阶段无法发现的 bug。要降低这些未知、不可预测的问题的影响，可以要求每个执行器周期性地到调度器那里进行核对，如果一些执行器跳过了过多的计划内核对，调度器就可以认为它已经 `LOST` 了。

假定某个 slave 和调度器之间的连接断开了，但是它和数据库的连接还在。该 slave 上运行的执行器还能够保持不中断，但是任务完成时调度器将无法接收到消息。这会导致管理员认为系统卡住了。

要解决这个问题，可以在执行器里添加心跳机制：执行器周期性地让调度器知道它还在运行，这样如果执行器丢失一些心跳之后，调度器会认为执行器 `LOST` 了，并且在一个健康的 slave 上重启任务。这也称为看护——不间断地主动监管，帮助快速检测到故障。自定义执行器可以使用 Mesos 消息 API 来实现心跳功能，确保整个系统保持健康。

Mesos 原生的心跳实现

Mesos 0.25 版本之后，会添加三种类型健康检查的原生支持——HTTP 请求、TCP 连接检查和运行命令。当这些都在 `TaskInfo` 里启用时，会发送周期性的 `StatusUpdates`。可以认为这就是 Marathon 提供的健康检查（详见“健康检查”部分）。Mesos 项目通常会将框架提供的有用功能集成回核心代码里。更新见 MESOS-2533 和 MESOS-3567。

增强的日志

自定义执行器的最后一个示例是增强 Mesos 的日志功能。Mesos 将每个进程的 `stdout` 和 `stderr` 写入执行器沙箱的本地文件。使用自定义执行器，可以将进程的 `stdout` 和 `stderr` 转发到中央化的日志存储库或者数据存储（比如 HDFS、S3 或者 Logstash）里，允许用户升级所有基于 `CommandExecutor` 的任务，中央化存储其日志。但是，还可以通过运行日志重定向器，比如将进程的 `stdout` 连接到 `syslog` 上的 `logger` 来实现，从而避免构建自定义的执行器。

重写 CommandExecutor

前面介绍了需要编写执行器的几种场景，本节详细介绍实际该如何做。从编写一些非常简单的执行器——和内建 `CommandExecutor` 兼容的执行器入手。

首先介绍 `MyExecutor` 的框架，它是 `CommandExecutor` 的克隆。示例 5-1 列出了所有导入和类的声明。

示例5-1 MyExecutor的导入和类声明

```
package com.example;
import org.apache.mesos.*;
import org.apache.mesos.Protos.*;
import java.util.*;
import java.io.File;
import org.json.*; ❶
import java.lang.ProcessBuilder.Redirect; ❷

public class MyExecutor implements Executor, Runnable { ❸
    // 之后会讨论这里的细节
}
```

❶ 使用 JSON 编码想要运行的任务。会从 `TaskInfo` 读入 JSON，这样就知道该调用什么。

❷ `ProcessBuilder` 是用 Java 启动进程的最为便捷的方式。使用它启动应用程序，并且从执行器将应用程序的 `stdout` 和 `stderr` 重定向到隔离的日志文件里。

❸ 因为没有查看进程什么时候结束的回调 API，所以这里启动一个线程等待进程的返回——这是实现 Runnable 的原因。

main 方法如示例 5-2 所示。

示例5-2 MyExecutor的main函数

102

```
public static void main(String ... args) throws Exception {
    Executor executor = new MyExecutor();
    ExecutorDriver driver = new MesosExecutorDriver(executor);
    driver.run();
}
```

虽然可以尝试通过命令行传入一些参数到 main 函数来配置一些东西，但是所有配置器的配置信息都来自于 Mesos API——这样通过确保所有配置都来自于单个源而简化了长期开发的过程。当调度器拥有执行器的完全控制时——就可以简化开发过程：全局（调度器）改动和本地（执行器）改动都在单个代码基——调度器的代码基里进行管理。这使得新特性的开发更为容易，并且简化了操作，因为管理员仅仅需要修改并且重新加载调度器，就能完成框架层的修改。

除了调度器，在执行器的 main 函数里创建了相应的 driver 和 Mesos 通信。不幸的是，该 driver 却是最麻烦的地方。当框架调用 MesosExecutorDriver.start() 时，需要已经由 slave 完成了启动。slave 注入一个环境变量 MESOS_SLAVE_PID，允许执行器连接到启动它的 slave 上，并且开始参与到 Mesos 的协议里。因此，如果想要测试执行器，需要实现一个模拟的 ExecutorDriver。如果忘记了这一点，不管开发执行器使用的是哪种语言，都会看到类似如下的诡异 stack trace：

```
F0605 21:32:01.538770 18480 os.hpp:173] Expecting 'MESOS_SLAVE_PID' in
environment variables
*** Check failure stack trace: ***
    @     0x7fc535912dfd  google::LogMessage::Fail()
    @     0x7fc535914c3d  google::LogMessage::SendToLog()
    @     0x7fc5359129ec  google::LogMessage::Flush()
    @     0x7fc535915539  google::LogMessageFatal::~LogMessageFatal()
    @     0x7fc5352c9ff0  os::getenv()
    @     0x7fc53534139b  mesos::MesosExecutorDriver::start()
    @     0x7fc5359068ee  Java_org_apache_mesos_MesosExecutorDriver_start
    @     0x7fc5390127f8  (unknown)
```

driver 完成之前都会阻塞主线程——当任务完成时，会在别的地方调用 driver.stop() 来确保执行器能够正确退出。因为执行器从容器及其运行的进程里解耦出了任务的概念，这一点很容易忘记，发送 TASK_FINISHED 并不会终止执行器。用户需要确保执行器完成工作之后会关闭其自身，或者确保调度器通过“杀死”其 canary 任务（详见“Canary 任务”部分）来显式地管理其生命周期。

103 和调度器类似，并不需要处理很多执行器的回调函数。示例 5-3 列出了目前可以忽略的回调函数。

示例5-3 可以忽略的回调函数

```
public void frameworkMessage(ExecutorDriver driver, byte[] data) { } ❶
public void registered(ExecutorDriver driver, ExecutorInfo executorInfo, ❷
                       FrameworkInfo frameworkInfo, SlaveInfo slaveInfo) {
    System.out.println("registered executor " + executorInfo);
}
public void disconnected(ExecutorDriver driver) { } ❸
public void shutdown(ExecutorDriver driver) { } ❹
public void reregistered(ExecutorDriver driver, SlaveInfo slaveInfo) { } ❺
public void error(ExecutorDriver driver, java.lang.String message) { } ❻
```

❶ 框架消息是调度器及其执行器之间通信的简单方式（但是并非完全保证，下面会详细介绍），通过发送任意序列化为 byte 的数据来完成通信。注意框架消息并不会路由到特定任务里。

❷ 当执行器首次连接到 slave 时调用该方法。最常见的做法是从 ExecutorInfo 里得到数据，因为可以在其 data 字段里存储执行器的配置信息。

❸ 当 slave 从执行器上断开连接时调用该方法，通常表明一个 slave 重启了。执行器很少需要在这里做什么特别的事情。

❹ 该回调函数通知执行器正常关闭。当 slave 的重启在规定时间内失败，或者执行器的框架完成时，会调用该函数。如果 5 秒（这是默认值，可以通过 slave 命令行参数 --executor_shutdown_grace_period 来配置）内关闭尚未结束，执行器就会被强行“杀死”。

❺ 当 slave 成功重启后会调用该回调函数，它包含新 slave 的信息。

❻ 当发生致命错误后调用该回调函数。当调用该回调函数时，driver 就不再运行。

执行器实际可以是 slave 检查点和恢复系统的完整参与者，但是，对于绝大多数执行器
104 而言，这一点可以忽略。之后本书会使用其中一些回调函数，但是现在，先探讨如何实际启动一个任务。

框架消息并不可靠

sendFrameworkMessage仅仅是libprocess语义的封装（详见“libprocess和actor模型”部分），因此其交付并不可靠。这意味着如果使用sendFrameworkMessage，框架不能依赖于消息的可靠交付——用户需要自行处理重试。但是这个不可靠的交付系统有什么用呢？虽然缺少完全保障，但是实际上除了集群出现特别极端的故障以外，这些消息都可以正常交付。这意味着对于所有公告性消息而言，sendFrameworkMessage是很好用的。

公告性消息的一种就是任务心跳。调度器想要知道消息是否还在继续进行，或者因为什么原因已经停止前进甚至死锁了。要检测到这些，会每分钟向调度器发送一个框架消息，让调度器跟踪5或10分钟还没有响应的执行器。如果因为临时原因消息没有交付，可以假定下一个消息很可能就会成功；但是，如果超过一段时间之后还没有消息的交付，这就意味着执行器真的有问题了（或者网络有问题）。本章“添加心跳”小结会详细介绍如何实现。

框架消息适用的另一个场景是提供UI里用户可能会使用的进度更新。有很多通信策略来确保如果仅仅丢失一次更新不会造成问题。比如，如果每次更新的是完成百分比，最新的更新会一直涵盖之前的更新（因为应该比之前的值大或者相等）。通常，框架消息很适合做新的更新代替之前更新的进度报告，而且这样的更新是供人使用，不是供程序使用的。

接下来探讨在执行器实现里需要访问什么数据。如下是MyExecutor类（见示例5-1）里定义的全局变量：

```
Process proc
```

需要存储所启动进程的引用，这样才能等待其完成。

```
TaskID taskId
```

还需要和Mesos沟通任务状态：和调度器通信的driver API要求TaskID作为参数。

```
ExecutorDriver driver
```

105

当然，想要使用driver，需要存储driver的引用。

工作正常的执行器必须让Mesos了解其任务的最新状态。因为发送状态更新有一些定式，

本书编写了一种便捷方法，如示例 5-4 所示。

示例5-4 状态更新帮助器

```
private void statusUpdate(TaskState state) {
    TaskStatus status = TaskStatus.newBuilder()
        .setTaskId(taskId)
        .setState(state)
        .build()
    driver.sendStatusUpdate(status);
}
```

这里简单构造了一个最基本的任务状态，并且通过 driver 发送出去。有时还会想要沟通一些额外信息。和大多数 Mesos protobuf 类似，TaskStatus 支持任意数据，以及人类可读的消息。

为了保持该示例的简捷性，执行器仅仅支持运行单个任务，运行后会退出。本书在帮助器功能里限制了这一点，见示例 5-5。

示例5-5 确保仅有单个任务

```
private boolean ensureOneLaunch(ExecutorDriver driver, TaskID id) { ❶
    if (this.taskId != null) { ❷
        TaskStatus status = TaskStatus.newBuilder()
            .setTaskId(id) ❸
            .setState(TaskState.TASK_ERROR)
            .setMessage("this executor only can run a single task") ❹
            .build();
        driver.sendStatusUpdate(status);
        return false;
    } else {
        return true;
    }
}
```

❶ 如果这是第一个任务就会返回 true，如果应该忽略就会返回 false。

❷ 启动任务的代码会设置 this.taskId 的值。因此可以假定如果已经设置了 this.taskId 的值，就已经启动了一个任务，因此是有效的。

❸ 注意使用新请求的任务的 TaskID，而不是正在运行的任务的。

❹ 在状态更新里利用人类可读的消息，来帮助用户诊断为什么任务启动导致某个错误。

示例 5-6 展示了实际如何启动一个进程。以配置的 JSON 对象为输入，并且返回新启动的进程。要帮助隔离日志，可以重定向任务进程的 stdout 和 stderr 到不同的文件，而不是留在执行器的 stdout 和 stderr 里。这里，任务 JSON 只有一个单键："cmd"，这是运行任务的命令行。可以简单地在 shell 里调用任务。

示例5-6 启动进程

```
private Process startProcess(JSONObject cfg) throws Exception{
    List<String> cmdArgs = Arrays.asList("bash", "-c", cfg.getString("cmd")); ❶
    ProcessBuilder pb = new ProcessBuilder(cmdArgs); ❷
    File stdoutFile = new File(System.getenv("MESOS_DIRECTORY"), "child_stdout"); ❸
    File stderrFile = new File(System.getenv("MESOS_DIRECTORY"), "child_stderr"); ❸
    pb.redirectOutput(Redirect.to(stdoutFile)); ❹
    pb.redirectError(Redirect.to(stderrFile)); ❹
    return pb.start();
}
```

❶ 构造要启动的进程的参数向量。

❷ ProcessBuilder API 是在 Java 里启动进程的简单方式。

❸ 执行器的所在路径由 slave 放置在环境变量 MESOS_DIRECTORY 里。最好将所有输出都放到这个目录下，因为当 slave 在执行器退出后需要重新声明磁盘空间时会对该目录自动进行垃圾回收。

❹ 重定向了任务进程的输出。

至此，终于可以处理任务的启动了。为此必须实现 Executor 接口的 launchTask 回调函数（见示例 5-7）。

示例5-7 实现launchTask

```
public void launchTask(ExecutorDriver driver, TaskInfo task) {
    synchronized (this) { ❶
        try {
            if (!ensureOneLaunch(driver, task.getTaskId())) { ❷
                return;
            }

            this.taskId = task.getTaskId(); ❸
            this.driver = driver; ❸

            statusUpdate(TaskState.TASK_STARTING); ❹

            byte[] taskData = task.getData().toByteArray(); ❺
            JSONObject cfg = new JSONObject(new String(taskData, "UTF-8"));
            this.proc = startProcess(cfg); ❸ ❻

            statusUpdate(TaskState.TASK_RUNNING); ❹

            Thread t = new Thread(this); ❼
            t.setDaemon(true);
            t.start();
        } catch (Exception e) {
            e.printStackTrace(); ❽
```

107

```
        }
    }
}
```

❶ 保护所有全局变量的访问，在执行器里的所有并发代码处都加上互斥锁。[1]

❷ 检查是不是接受到的第一个任务。如果不是，什么也不做，直接返回。

❸ 如前所述，将这个回调函数能访问到的信息存储到全局变量里：TaskID，ExecutorDriver 和 Process。

❹ 随着任务处理的进行，持续通知 Mesos 进度。

❺ 必须提取并且解析任务的细节。

❻ 最后启动进程。

❼ 需要监控进程，了解何时结束，以及是否成功结束。下文会详细介绍这个监控进程的实现。

❽ 在实际生产环境代码里，必须能够容易地诊断执行代码的哪部分出错了。如果异常是致命的，通过 StatusUpdate 传递异常是最容易的实现方式。

108 至此已经基本了解了执行器如何管理其任务的生命周期。示例 5-8 展示了如何跟踪任务的完成情况。

示例5-8 等待进程结束

```
public void run() {
    int exitCode;
    try {
        exitCode = proc.waitFor(); ❶
    } catch (Exception e) {
        exitCode = -99; ❷
    }
    synchronized (this) { ❸
        if (proc == null) { ❹
            return;
        }
        proc = null; ❺
        if (exitCode == 0) { ❻
            statusUpdate(TaskState.TASK_FINISHED);
        } else { ❼
            driver.sendStatusUpdate(TaskStatus.newBuilder()
                    .setTaskId(taskId)
                    .setState(TaskState.TASK_FAILED)
```

1 Java 同步知识不在本书范围之内，就相信这样做是正确的吧。

```
                .setMessage("Process exited with code " + exitCode)
                .build());
        }
    }
    driver.stop(); ❽
}
```

❶ waitFor() 会阻塞到进程终止为止，并且返回进程的退出码。

❷ 如果因为某些原因 waitFor() 抛出异常，那么假定发生了严重错误，并且使用特别的退出码标记这一情况。

❸ 如前所述，this 上的互斥锁可以保护全局变量的访问。

❹ 如果执行到这里，并且 proc 是 null，这是一个特别的标记，意味着进程被 killTask “杀死”了（这种情况笔者还没遇到过）。这样的话，工作已经完成了。

❺ 将 proc 设置为 null，从而设置上特别的信号。

❻ 当进程带着成功状态退出时，报告这个任务完成了（记住，FINISHED 是 Mesos 里的成功状态）。

❼ 否则，表明这个任务失败了，因此发送 FAILED 状态。

109

❽ 这里 stop() 执行器的 driver，这样 main 函数能够返回，并且执行器能够正常关闭。

前文已经讨论过新的执行器如何处理接受任务、启动进程以及当进程完成时发送合适的状态更新。还缺失的部分是处理来自调度器的请求，提前“杀死”任务。示例 5-9 展示了如何处理 killTask 消息。

示例5-9 killTask的实现

```
public void killTask(ExecutorDriver driver, TaskID taskId) {
    synchronized (this) { ❶
        if (proc != null ❷
                && taskId.equals(this.taskId)) { ❸
            proc.destroy(); ❹
            statusUpdate(TaskState.TASK_KILLED); ❺
            proc = null;
        }
        driver.stop();
    }
}
```

❶ 如前所述，必须使用互斥锁保护全局变量的访问。

❷ 如果 proc 是 null，那么进程已经消亡了，或者因为它结束了，或者因为提前

killTask 了。

❸ 还必须确保运行着的任务是应该被“杀死”的任务。单任务的执行器并非严格要求这一点，但是它能够帮助避免调度器内的编程错误，以防向错误的执行器发送“杀死”消息。

❹“杀死”进程。

❺ 通知 Mesos 按照用户的请求，任务已经被“杀死”。注意如果在调用 destory() 后但是在发送状态更新前线程崩溃的话，就可能导致竞争：Mesos 可能还认为任务仍然在运行。这也是之后将 proc 设置为 null 的原因——它会确保来自调度器的第二次“杀死”消息会重新发送 TASK_KILLED 消息。

至此大家应该已经知道如何实现一个基础的，功能类似于 CommandExecutor 的执行器了。

当然，你可能想知道从调度器处实际如何使用执行器。要将该执行器和第 4 章的示例调度器集成，只需要简单更新 Job 对象的 makeTask 方法，如示例 5-10 所示。

示例5-10 新执行器的加强makeTask

```
public TaskInfo makeTask(SlaveID targetSlave, FrameworkID fid) {
    TaskID id = TaskID.newBuilder()
        .setValue(this.id)
        .build();
    ExecutorID eid = ExecutorID.newBuilder() ❶
        .setValue(this.id)
        .build();
    CommandInfo ci = CommandInfo.newBuilder()
        .setValue("java -jar /path/to/custom/executor.jar") ❷
        .build();
    ExecutorInfo executor = ExecutorInfo.newBuilder() ❸
        .setExecutorId(eid)
        .setFrameworkId(fid)
        .setCommand(ci)
        .build();
    JSONObject cfg = new JSONObject();
    try {
        cfg.put("cmd", this.command);
        return TaskInfo.newBuilder()
            // 省略未改动代码                    ❹
            .setExecutor(executor) ❺
            .setData(ByteString.copyFrom(cfg.toString().getBytes("UTF-8"))) ❻
            .build();
    } catch (Exception e) {
        e.printStackTrace();
        throw new RuntimeException();
    }
}
```

❶ 显式分配给执行器一个 ID。CommandExecutor 为任务和执行器使用相同的 ID，因此本书示例也这么做。

❷ 需要构建自定义执行器的 .jar，并且将其分发到 slave 上。下一节会仔细介绍这里的可用解决方案的细节。

❸ ExecutorInfo 要求几类信息，包括 FrameworkID。调度器会将其传递给 makeTask。

❹ 这里没有重复展示未改动的 TaskInfo 配置（详见示例 4-3）。但是，确实删除了 .setCommand()，因为这里正在配置执行器。

❺ TaskInfo 必须带有命令行或者执行器集，但是永远不能同时带有这两者——这样会导致错误。 111

❻ 最后，包含进任务的 JSON 配置。

所有工作都完成了！我们完成了第 4 章调度器的加强，让其支持新的，自定义的调度器。

引导执行器的安装

创建自己的执行器的挑战之一是需要在所有地方部署执行器的二进制文件。本节介绍解决该问题的四大方案，并且评估每种方案的优势和劣势。

使用 HDFS、S3 或者其他非 POSIX 存储

任何 Mesos 的执行器都能列出将要下载的 URI 集合，还可以选择在执行器启动之前解压到本地工作目录。Mesos 内建支持 HTTP、FTP 和 HDFS 存储来下载文件（详见“配置进程环境”部分）。因此，可以在本地 HDFS 集群、Amazon S3 或者其他数据存储上轻松托管应用程序，避免部署任何其他策略可能会要求的新技术。该方案的挑战是数据存储必须是充分可扩展的，能够交付容器给每个并发启动的任务。新开发的框架通不过可扩展测试的一个常见原因是，它同时启动了上百个执行器，这些执行器全都同时向单个服务器请求二进制文件，从而导致服务器崩溃。这个问题也称为惊群问题（Thundering herd problem）。用户需要确保能够充分复制提供这些二进制文件的服务器，或者使用可扩展性强悍的系统，比如 S3。

该方案的“下一级别”版本是在调度器进程里实际托管二进制文件，让调度器运行一个嵌入式 HTTP 服务器。当然，这会加剧惊群问题。解决方案是让调度器错开任务的启动时间，确保服务器永远不会压力过大。将启动命令提交给 Mesos 之前随机等待一段时间可以保证启动时间错开。

很多 Mesos 集群都是使用配置管理系统来完成部署的，比如 Chef、Puppet 或者 Ansible。如果这适用于你的集群，你就可以使用这些配置管理系统将执行器二进制文件推送到 Mesos 集群里的每个 slave 上。这么做的好处是可以确保这些二进制文件一直处于立即可用于启动任务的状态——无须等待其下载或者担心网络故障导致任务变成 LOST 状态。但是，缺点是部署以及升级到执行器的新版本会更加困难。当然，还需要版本化部署了的每个执行器（确保升级期间多个版本能够同时运行）。然而，在部署阶段要等待执行器推送到所有 slave 上，这会非常枯燥。

使用共享的 POSIX 文件系统

通常，计算集群已经有了共享的和 POSIX 兼容的文件系统。流行的选择有 NFS 和 GlusterFS。共享文件系统的方案类似于配置管理方案：在这两种场景下，执行器都存在于已知路径下。将该路径放在共享文件系统上的好处是用户无须等待配置管理系统将二进制文件实际推送到每个 slave 上，相反，可以在执行器拷贝到共享文件系统上时就立即启动执行器。缺点是很难扩展共享文件系统，如果它发生故障，整个集群就会瘫痪。

使用 Docker

Mesos 很好地支持了 Docker 容器（详见“使用 Docker”部分），Docker 容器提供了交付完整二进制数据镜像的方案。一旦 Docker 化执行器，就可以在集群里任意 slave 上启动该容器。这里的挑战是 Docker 的存储库：必须找到能够托管 Docker 容器的 Docker 存储库（可以是 Docker Hub 或者自己管理的存储库），需要可充分扩展并且足够安全。该方案的缺点是下载 Docker 容器的时间太长：当 Docker 容器需要 10 分钟才能下载时，执行器的启动很可能已经超时了。这会导致不可预测的，短暂出现的 LOST 任务。

上文已经介绍了可选方案，那么该如何从其中选择出正确的方案呢？如果你已经在重度使用 Docker，那么推荐继续使用 Docker 托管 Mesos 执行器。否则，如果已经有了共享的 POSIX 文件系统，那么 POSIX 的方案提供了最快最简单的部署方式。如果不想为此搭建共享 POSIX 文件系统，那么在 HDFS 或者 S3 上提供二进制文件也能完成快速部署，并且几乎所有人都能够完成这里所需的工作。使用配置管理系统的确能够将部署中央化，但是这么做会大幅降低重新部署更新或者更旧版本执行器的速度，因此这是最后推荐的方案。

Fetcher 缓存

Mesos 0.23 里发布了称为 fetcher 缓存的新功能。fetcher 缓存意图降低 S3、HDFS 或者其他文件系统的工作负载，通过在每个 slave 上缓存执行器的下载来实现，这样每个 slave 只需要下载一次即可。虽然这并没有彻底消除共享文件系统的可扩展问题，但是的确很有帮助。要充分利用 fetcher 缓存改进可扩展性，还需要确保调度器错开执行器的启动。这样，共享文件系统永远不会有太多并发的请求，fetcher 缓存会进一步降低工作负载。fetcher 缓存确保每个 artifact 在每个 slave 上只会下载一次，即使多个执行器请求同一个 artifact，也只需等待单次下载完成即可。

要使用 fetcher 缓存，只需在 `TaskInfo` 里的 URI protobuf 里简单地将 `cache` 设置为 `true` 即可。fetcher 缓存通过用户隔离，因为它只是简单地使用 `URI` 作为键。以后，可能会提供绕过或者清理缓存的方法，但是现在，对于资源内容的每个不同版本而言，URI 都必须是唯一的。

在任何情况下，都要记住调度器决定从哪里以及如何获取并且启动执行器。因此，调度器的每个版本都能够选择启动执行器的不同的向前以及 / 或者向后兼容的版本。确保调度器永远能够启动可兼容的执行器，可以避免框架内内部版本的不匹配。

添加心跳

Mesos 架构的一大优势是临时的通信故障不会影响到运行着的执行器的执行。通常在设计高可用时需要利用这个功能，因为即使发生其他故障，每个执行器都还能够继续处理工作或者服务客户端的请求。当构建服务器池调度器时（详见“服务器池调度器”部分），这正是我们需要的行为。另一方面，有时候想要确保每个执行器都正在运行，以便能够快速处理故障——毕竟，没有方法能够区分开临时的和永久的网络波动。[1]

通过要求每个执行器间隔发出心跳（这里的间隔取决于对故障处理时效的要求），来确保每个执行器都在正常工作。使用框架消息发出心跳，这里有几个有意思的特性和注意事项：

- 框架消息不可靠：是否交付没有绝对的保障。
- 框架消息可扩展：它们通常使用从执行器到调度器的优化了的传输机制，从而避免 master 的瓶颈。
- 框架消息能够带有任意二进制数据负载：可以序列化想要包含进消息的任何数据，比如进度更新。
- 框架消息在执行器和调度器之间：如果想要发送执行器上某个特定任务的消息，

1 这称为 FLP 不可能的结果。

就可以在二进制负载里包含任务 ID。

首先需要在执行器里添加心跳，如示例 5-11 所示。

示例5-11 执行器里的心跳

```
public static void main(String ... args) {
    // 省略无关代码
    Timer timer = new Timer(); ❶
    timer.schedule(new TimerTask() {
        public void run() {
            driver.sendFrameworkMessage(new byte[1]); ❷
        }
    }, 0, 5000); ❸
    // 省略无关代码
}
```

❶ 创建一个新的 Timer，在执行器启动时管理心跳。

❷ 创建的任务简单发送一个不包含什么数据的框架消息。在生产环境框架里，可以在这里包含一些有用的信息，比如发送消息的时间（用来跟踪消息交付的延时），或者每个执行器任务的完成百分比是多少。

❸ 立即启动该任务，每 5 秒发送一次心跳。要提高可扩展性，通常需要每隔 30 秒或者几分钟发送一次心跳。

虽然在执行器里发送心跳消息很容易，用户还是需要增强调度器。首先，加强 Job，当其没有及时收到心跳消息时会自动失败，如示例 5-12 所示。

115

示例5-12 自动销毁自身的Job

```
public class Job {
    private Timer timer = new Timer(); ❶
    private TimerTask missedHeartbeatTask; ❷
    // 省略无关代码

    public void heartbeat() { ❸
        if (missedHeartbeatTask != null) { ❹
            missedHeartbeatTask.cancel();
        }
        missedHeartbeatTask = new HeartbeatTask();
        timer.schedule(missedHeartbeatTask, 20000); ❺
    }

    public static Job fromJSON(JSONObject obj) {
        Job job = new Job();
        // 省略无关代码
        job.heartbeat(); ❻
        return job;
    }
```

```
    public void succeed() {
        省略无关代码
        missedHeartbeatTask.cancel(); ❼
    }

    public void fail() {
        省略无关代码
        missedHeartbeatTask.cancel();
    }

    private class HeartbeatTask extends TimerTask { ❽
        public void run() {
            System.out.println("Heartbeat missed; failing");
            fail(); ❾
        }
    }
}
```

❶ 每个 Job 都有一个 Timer 管理其最新心跳的过期时间。

❷ 必须保存会导致 Job 的当前实例失败的 TimerTask，这样在接收到心跳消息时可以将其取消。

❸ 每次从执行器接收到心跳消息时都会调用 heartbeat() 方法。

❹ 每次接受到心跳消息后，必须取消当前任务，因为如果这个任务运行的话，Job 就会失败。

❺ 当接受到心跳消息后，如果 20 秒内没有收到下一个心跳消息就安排一次失败。因为预计每隔 5 秒会收到一次心跳消息，当丢失好几次连续心跳后才会运行这段代码。 <116>

❻ 第一次创建 Job 时，向其发送一次心跳消息，确保如果执行器不开始运行的话，Job 会失败。

❼ 当任务完成时，取消待执行的失败任务，因为不再需要这些任务了。

❽ HeartbeatTask 是 TimerTask 的子类。仅仅需要实现 run() 方法，只有超时前任务没有被取消时才会调用该方法。

❾ 如果任务的确开始运行了，仅需要 fail() 该 Job，这样之后就会重新调度这个 Job。

至此，已经加强了 Job 来跟踪心跳，还需要做的是将框架消息从调度器传递到合适的 Job 里。要达到这一目的，需要在调度器里实现 frameworkMessage 回调函数，如示例 5-13 所示。

示例5-13 为心跳实现frameworkMessage

```
public void frameworkMessage(SchedulerDriver driver, ExecutorID executorId,
                             SlaveID slaveId, byte[] data) {
    String id = executorId.getValue(); ❶
    synchronized (jobs) {
        for (Job j : jobs) { ❷
            if (j.getId().equals(id)) {
                j.heartbeat(); ❸
            }
        }
    }
}
```

❶ 对于本章的示例执行器而言（对于 CommandExecutor），将执行器 ID 设置等同于作业 ID 和任务 ID。

❷ 因为该示例不关心框架消息里的数据，所以简单地扫描所有当前作业，查清是否其中某一个和刚发送心跳消息的执行器带有相同的 ID。

❸ 一旦找到匹配项，调用 Job 的 heartbeat() 方法，这样示例 5-12 里的代码就能够完成其工作了。

117 这就向框架里添加了基本的心跳功能。当然，对于符合生产环境质量的系统而言，还有一些需要加强的地方。首先，这里展示的心跳机制为每个 Job 的心跳创建了一个计时器。对于有上百个 Job 的系统来说，每个 Job 可以共享相同的 Timer 来确保可扩展性，因为每个 Timer 会创建一个新线程。其次，该心跳系统并不会真的“杀死”没有心跳的 Job，它只是将其标注为死锁。应该使用 killTask API“杀死”不发出心跳信息的执行器，否则就可能会导致孤儿或者僵尸执行器，它们会消耗集群资源却不做什么有用的工作。最后，当不确定某个执行器是否被正常“杀死”时，要很小心地防止在某个 slave 上启动带有重复 ID 的执行器。必须跟踪所有使用了的执行器 ID，为每个执行器生成新的、唯一的 ID。使用上文代码时，当之前的执行器还在运行但是却因为缺少心跳被认为不活动时，如果尝试在相同 slave 上重新启动该 Job，就会导致异常行为。

至此，本章已经介绍了通过使用任务级别和执行器级别的 API，如何构造出简单的执行器。还添加了一些内建执行器所不具备的额外功能，比如心跳。接下来的几节会介绍如何向执行器添加其他高级特性。

高级执行器特性

到这里，你很可能意识到编写执行器很困难。不像其他大多数软件，执行器特别难测试，因为需要跨集群构建、打包和交付，某些地方还必须等待，检查其是否能够和调度器正

确交互。

向执行器添加额外特性之前，一定要记住执行器并不一定要将其任务作为子进程。实际上，经常在执行器的相同进程中运行任务——这样做效率会更高，设计也更为简单。将执行器和任务分隔到不同进程中的唯一原因是，这样才能模仿 Mesos 里的内建 `CommandExecutor` 的行为。

进度报告

“添加心跳”一节简要介绍了这个概念，额外使用周期性框架消息来监测执行器是否还在运行并且可访问，可以使这些消息里包含周期性状态更新信息。但是，并不是一定要用框架消息才能完成这个功能：`TaskStatus` 消息也有一个 `data` 字段，也可以使用这个字段发送任务相关的更新。此外，还可以为相同的状态（比如 `TASK_RUNNING`、`TASK_STARTING` 等）发送多个 `TaskStatus` 更新，但是 `data` 字段带有不同的值来可靠地发送 118 任务的内部状态更新。比如，可能希望任务开始初始化自身时，使用值为“initializing”的 `data` 来报告它在 `RUNNING`，当任务准备好开始处理请求时，使用值为“ready”的 `data` 来报告它在 `RUNNING`。那么基于什么来决定是用状态消息还是框架更新来做进度更新通信呢？

框架消息是高度可扩展的，因为它们直接从执行器发送到调度器。因为 Mesos master 通常并不处理或者转发框架消息。[1] 这些消息的吞吐量受限于调度器消费消息的能力。此外，因为框架消息并非完全可靠，高负载时 Mesos 可以选择丢弃一些消息，从而确保高优先级的消息会被处理。这些特性使得，在可扩展性最重要时框架消息是最好的选择。

当然，可扩展性是有代价的：如上所述，有时候框架消息会被丢弃。而另一方面，状态更新能够确保交付。但是这样的保障会导致巨大的性能开销：每次发送状态更新前都必须创建检查点到 slave 的磁盘里，随后当更新成功发送之后需要第二次磁盘写入来标记此次更新。此外，slave 必须跟踪所有重要的状态更新，这样当状态更新应该被交付时如果调度器临时掉线了，之后 slave 还能够重新发送该状态更新。这样的检查点和跟踪，以及周期性重新发送未答复的状态更新，共同保障了状态更新的可靠性，使得它很适合向调度器通知任务里重要的生命周期事件，所付出的代价则是多得多的资源使用量，以及 master 和 slave 的磁盘可能会出现的瓶颈。

因此，框架消息还是状态更新更合适的关键指标是，是否每次更新完全覆盖之前的所有更新。比如，如果目标是报告任务完成的百分比，那么遗漏了 49% 的更新并不重要，因为很快会接受到 50% 的更新（50% 的更新表明已经达到了 49%）。这时，框架消息更为

1 从技术上讲，Mesos master 的确有时会处理框架消息。这会在特殊情况下导致瓶颈，但是绝大多数时候使用的是优化了的传递型 master。

合适。另一方面，如果任务要报告的是多个子任务是否已经完成，那么每次更新都是独一无二的，它们都必须交付给调度器。这时，状态更新则更加合适。

119

添加远程日志

构建分布式系统常会遇到的问题之一是搞清楚“正在发生什么”这一点通常非常困难，因为日志分散在集群里。在其中搜索错误消息很困难，因为需要搜索整个集群的日志。此外，通常还需要关联一些日志来确定集群内多个程序之间如何交付会导致故障。最常见的解决办法是将所有日志发送到中央化的存储库里，之后能够辅助用户分析，搜索，以及关联日志。实现这一方案的流行技术是 Splunk、Logstash 和 Papertrail。[1]

还记得如何使用 `ProcessBuilder.redirectOutput()` 将进程的 `stdout` 和 `stderr` 发送到文件里吧——还可以使用直接来自于进程的 `InputStream`，这样可以将输出发送到任何地方。一旦开始编写自己的执行器，就能够完全控制执行器日志发送到何处，以及任务日志发送到何处。对于很多应用程序而言，因为每个执行器只有一个任务，会将执行器和任务的日志发送到相同的地方。但是，对于另外的应用程序而言，实际上可以将任务和执行器的日志发送到不同的地方。这在想要在和其他任务隔离的情况下调试某个任务时很有用处，我们不想看到所有执行器任务的日志都搅和在一起。

多个任务

Mesos 执行器设计上可以用来运行多个任务。在已有执行器上运行一个任务，像之前那样正常启动该任务即可，除非需要使用已经在那台 slave 上运行着的相同的 `ExecutorInfo` 作为执行器。为了辅助执行器的重用，每个 offer 有一个名为 `executor_ids` 的字段，它包含当前在这台 slave 上运行的所有执行器的 `ExecutorID`；但是，还得依赖于用户将所有 `ExecutorInfo` 存储起来，并且在任务描述符里包含这些信息。master 会验证 `TaskInfo`，因此如果所提供的 `ExecutorInfo` 和想要在其上运行的执行器不匹配，任务会立即失败并标记为 `TASK_ERROR`。

120

改变容器的大小

执行器的一个强大特性是其大小永远是它们运行的所有任务和执行器的资源之和。自定义执行器是能够根据所执行任务轻松地动态调整容器大小的唯一方式。当然，天下没有免费的午餐：用户需要持续跟踪每个执行器运行在哪个 slave 上，并且当想要扩大容器时，需要使用来自已有执行器的 slave 的 offer。

1 Jason Wilder 的博客里有中央化日志解决方案的总结和对比。

带有昂贵状态的执行器

通常，当执行器包含某类构造很昂贵的状态时就完成了执行器的重用。比如，执行器会维护存储在外部慢速文件系统，比如 S3 里的大量数据的本地缓存。这时，执行器负责下载数据，随后每个任务就能够像访问本地文件那样访问数据，这比跨网络读取数据要快得多。当然，永远需要权衡：如果同时在该执行器上运行多个任务，这些任务就会运行在相同的容器内——这也就放弃了 Mesos 所提供的任务间隔离的保障！如果用户能够确保任务不会意外占用比所需更多的容器资源，这也是一种可行的方案。但是，如果你也属于绝大多数可能会制造 bug 的开发人员，那么这同时也是一种风险较高的方案，因为它让框架可能面临预计外的干扰和可能的不确定性。

多阶段初始化

在相同执行器上使用多个任务时，不太常见的一种情况是执行器的初始化过程非常复杂。不是如“进度报告”小节所建议的，在初始化发生时发送状态更新，调度器可能认为初始化的每一步都是单独的任务。这样的话，可以在调度器上管理初始化逻辑，而不是在执行器上。虽然偶尔能看到使用这种技术，但是本书建议不要这么使用。在执行器里实现初始化状态机要更为容易。不要在执行器里实现状态机的一部分，而在调度器里实现另外一部分，这样会增加执行器初始化阶段框架对网络可靠性的依赖程度。

Canary 任务

但是，还有一种情况下完全有必要在执行器上附加额外任务：当前没有运行任何任务（但是在等待新任务）的执行器关闭或者崩溃时，用户需要能够接收到及时的通知。如之前 `Scheduler` 回调函数（见示例 4-6）所述，你认为可能会在执行器丢失时给出通知的回调函数其实永远也不会被调用。因此，要想检测到执行器的消失，需要在其上运行一个 canary 任务。这样，当执行器关闭、崩溃或者其 slave 失活时，就会通过 canary 任务接收到 `TASK_LOST`，告诉用户执行器丢失了。当然，如果不需要及时检测到执行器的丢失，可以当一定时间内没有收到来自某个执行器的 offer 时，就假定它失活了。

121

什么时候应该使用它们

执行器启动多个任务的功能并不常用，这是有一定原因的。这里的 API 很难用，因为需要保存 ExecutorInfo 以供重用。因此，一般只在如下情况下才考虑使用多个任务：

- 有必要随执行器工作负载的变化而动态改变其资源总量时。
- 当执行器状态的构造本身非常昂贵，从而值得抛弃 Mesos 的隔离功能来节省重新初始化的消耗时。
- 当必须快速检测到可能没有运行任何任务的执行器的故障时。

要小心使用该特性，因为它会让调试变得非常困难。

本章小结

执行器是框架 worker 的 Mesos 抽象。当框架要求调度器和 worker 之间频繁交互时就应该使用执行器。构建执行器通常很困难，因为需要在开发中频繁向集群部署新 build。为了克服这些困难，可以构造自定义执行器，使得框架能够动态扩展在其上运行任务的容器，高效共享昂贵的资源，并且提供易用的 API 在调度器和 worker 之间发送消息。

本章首先探讨了可能需要构造执行器的原因，然后实现了和内建 Mesos CommandExecutor 兼容的执行器，并且和第 4 章介绍的作业调度器集成。之后添加了心跳功能，该功能帮助系统更快地检测到集群上某些类型的故障，给需要等待作业完成的用户带来更好的体验。

最后，讨论了自定义执行器的一些高级用法：如何在框架消息和状态更新之间做选择，和中央化分布式日志系统集成时的思考，以及如何和为什么开发支持多任务的执行器。

122 至此，本书已经介绍了在 Mesos 上构造应用程序的完整知识体系——使用已有框架，构建应用程序特定的调度器，创建功能丰富的执行器。后面的章节会继续探讨更加高级的话题，比如 Mesos 内部机制、Docker 集成，以及外部、先进的 API。

第 6 章

123

Mesos的进阶主题

虽然前面已经学习了在 Mesos 上构造应用程序的方方面面，但是你的问题可能比一开始还要多：

- Mesos 的内部架构是怎样的？
- Mesos 如何处理故障？
- 怎么在 Mesos 上使用 Docker？

本章会介绍所有这些主题，这样就能够构造出更加精妙、可靠的系统了。

libprocess 和 actor 模型

从高层看，Mesos 是一个混合的强一致性 / 最终一致性的系统，基于消息传递 actor 模型，用 C++ 编写。

actor 模型是编程实现同步分布式系统的范式。在 actor 模型里，开发人员实现了 process 或者 actor，这是一次处理一条消息的单线程代码模块。当处理消息时，actor 可以向其他 actor 发送消息或者创建新的 actor。actor 只能向它知道进程标识符（PID）的进程发送消息。当一个 actor 生成另一个 actor 时，它能够知道新创建 actor 的 PID。为了得到更多 PID，actor 必须在消息里互相发送 PID。

Mesos 使用的 actor 模型框架称为 libprocess，由 Ben Hindman 在编写 Mesos 的同时完成。[1]libprocess 是一个 C++ API，提供 actor 模型的语义，并且使用 HTTP 通信。因为所有程序基本都能理解 HTTP，因此很容易和 libprocess 连接。

124

libprocess 消息是 HTTP 上的一种 protobuf，可以受益于头消息、内容谈判，以及其中的

1 它作为 Lithe 项目的一个组件出现。

路由特性。libprocess 进程由其 PID 唯一标识，是主机名和 actor 端口。

有趣的事实

为了支持 actor 模型所必需的异步语义，libprocess 使用了一些技巧：它认为 HTTP 状态码 202 意味着请求已经被接受，并且正在被异步处理。

对于 Mesos 的大部分用户而言，并不需要理解 libprocess 和 actor 的任何细节。但是，如果用户想要使用高级容器化技术（比如 Docker 桥接网络）或者想要有多个网络接口，那么就需要为 Mesos master 和 slave 配置 `LIBPROCESS_IP` 和 `LIBPROCESS_PORT` 环境变量。这些环境变量允许用户控制 master、slave 和其他所绑定的 libprocess actor 使用什么网络接口，从而保证它们的消息能够成功到达目的地。

一致性模型

本节介绍 Mesos 的一致性模型。它依赖于 CAP 定理的哪个部分呢？[1] 作为 Mesos 的用户和开发人员，什么能保证我们能够得到想要的东西？这个问题并不好回答。本节会详细介绍带有三个 master 的 Mesos 集群，该集群启用了 registry 特性，以及 slave 的检查点和恢复特性。实际上，这里假定启用了所有和可靠性、持久化相关的特性，因为这些特性构成了 Mesos 的健壮一致性模型的基础。

图 6-1 是 Mesos 块状图，其中一个框架运行着一个任务。这里有四大部分：master、slave、框架的调度器和框架的执行器。有三条通信链路：master 和调度器之间、master 和 slave 之间，以及 slave 和执行器之间。这里依次分析它们的故障模型。

125

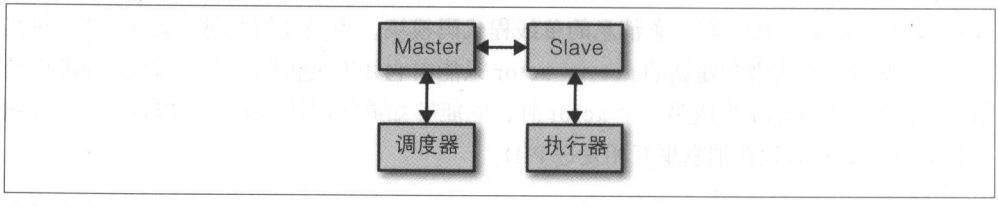


图6-1 框架的消息流

1 CAP 定理是关于分布式系统可能以及不可能行为的理论性结论。详见 Seth Gilbert 和 Nancy Lynch 所著的 *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services*。

如何处理 slave 的故障

首先一起看看 slave 的生命周期，以及多种故障是如何处理的。最初，当新的 slave 启动时，它会连接到 Mesos，向 master 发送 `Register` 消息来请求注册。master 随后在 registry 里保存 slave 的信息，一旦该操作提交了，就会向 slave 发送 `Registered` 消息，通知 slave 它已经成功加入了 Mesos 集群。这时，master 就可以开始向连接着的调度器提供该 slave 的资源。

假定这时候 slave 崩溃了，master 会立即停止接受 slave 的心跳，并且当达到配置的超时时间后，slave 会被认为已经“死”了，并被移出 registry。因此，即使 slave 只是暂时断网，一旦它恢复心跳或者尝试重新连接到 master，master 也会礼貌地通知 slave 将自身“杀死”。[1] 这里，最好在监管程序，比如 Supervisord、Monit 或者 SystemD 的监管下运行 slave。当 slave 重启时，会为自己生成一个新的随机 slave ID，这时就能够作为新的 slave 成功注册了。当 slave 崩溃时，这个 slave 上运行的任何任务都会 LOST，因为 slave 已经“死”了。

故障停止 vs. 故障恢复

当学术研究员们设计新的健壮分布式算法时，他们通常会致力于两种不同的模型：故障停止和故障恢复。在故障停止模型里，无论何时，机器崩溃了就会永远保持崩溃状态，这台机器永远也不会恢复正常。在故障恢复模型里，允许崩溃的机器恢复正常。可以看出，故障恢复模型更贴近现实——毕竟，当故障发生时，我们会修复并且重启机器，而不会在机器第一次出问题时就直接将它们丢进垃圾箱（或者丢给供应商）！不幸的是，事实证明故障恢复模型非常难以证明算法是否真的有效，因此大部分研究都是在故障停止模型下完成的。有意思的是，Mesos slave 使用了巧妙的方式模拟故障停止模型——当启动一台 slave 时，其 ID 并不仅仅是运行着的机器的主机名，而且是唯一生成的。这意味着如果一台 slave 崩溃并且随后重启了，它会得到一个新的 slave ID，从而对 Mesos 而言它就是一台新的 slave 机器。这样的设计使得用户更容易理解 Mesos 的行为。需要注意的是，如果需要，Mesos 的检查点特性可以让重启的 slave 表现为旧版本。在这种情况下，我们认为 slave 的重启就和短暂的网络问题一样，而不是一次崩溃。

126

1 注意如果 slave 在配置的超时时间内重新连接，slave 及其所有任务都可以继续正常工作。该行为也正是 Mesos 的零下线时间实时升级所依赖的。

如何处理 master（或者 registry）的故障

本节讨论如果 master 崩溃会发生什么。首先，需要知道这不是什么大事，因为 Mesos 构建时就考虑到了各种类型分布式故障的处理。另外，我们也搭建了冗余的备份 master，不是吗？当之前的 master 崩溃后，会“选举”产生新的主 master。slave 会逐渐通过自身和 master 的定期通信了解到 master 的更新信息。因为 slave 会周期性发出心跳，它们会看到其心跳的连接失败或者被拒绝（取决于旧的主实例是关闭了还是仅仅重启并且成为了备份 master）。slave 会再次查询 ZooKeeper 得到新 master 的信息。这时，slave 会使用不同的 RPC 连接到新的 master 上，称为 `Reregister`，因为 slave 认为自身已经注册了但是无法连接。当新的主实例接收到 RPC 之后，它会检查 registry 确认这个 slave 是否有效。如果是，就会接受 slave 的注册请求。但是，如果该 slave 不在 registry 里，那么它会告诉 slave 将自身“杀死”，这样 slave 就能够获得一个全新的 slave ID，并且作为没有任何历史的新 slave 重新加入集群。

到这里，你可能会问为什么需要 registry 和这样的重新注册流程呢。让我们一起看看如果移除了重新注册和 registry 之后会发生什么。重新注册是一种简单的方式：如果一个 slave 无法重新注册，那么 master 崩溃以及后续故障转移时，会一直强制该 slave 作为新 slave 注册，因为（如前面所述）Mesos 使用的是故障停止模型。这也意味着会“杀死”该 slave 上的所有任务。不用说，当 master 需要重启或者故障转移时，立即“杀死”整个集群内的所有任务，这样的设计很不好，这和使用高可用性集群操作系统的初衷是背离的。

现在，思考一下如果移除了 registry 会发生什么。假定 slave 因为网络问题停止了心跳，并且旧的主实例认为该 slave 已经 dead 了，这也意味着其上运行的所有任务都被认为 `LOST` 了。随后，master 故障转移到一个新的实例上，这时那个之前没有发送心跳的 slave，奇迹般地再次开始发送心跳消息了。现在，新的主实例会接收到来自该 slave 的重新注册请求并且接受该请求。但是等一下！旧的主实例认为该 slave 已经 dead，并且给调度器发送了该 slave 上所有任务都 `LOST` 了的消息。因此系统会进入一个糟糕的状
127 态，在 `LOST` 状态的任务奇迹般地复活了。这是一个 bug，因为 `LOST` 是任务的终止状态。registry 帮助用户确保任务无法从 `LOST` 状态（或者任何终止状态）转变回非终止状态：旧的主实例在 registry 里将 slave 标记为 dead，这样新的主实例就能够知道应该拒绝该 slave 的重新注册请求。

这时你可能会想，“似乎 master 不得不做大量记录，来跟踪每个 slave 上的每个任务都处于什么状态，并且将这些都保存到 registry 里。”这么理解可以说是对的，除了 master 实际上并不需要保存任何任务信息到 registry 里。因为 registry 让用户知道哪些 slave 是有效的，Mesos 认为这些 slave 是其任务的可信的、权威的信息来源。因此，当 master 故

障转移发生时，新的主实例并没有哪些任务正在运行的信息（实际上，Mesos UI 能够反映这些信息，因此 master 故障转移之后要过几分钟才能再次看到任务列表）。随着时间的消逝，因为 slave 重新注册并且报告，master 就能够重新构建出与集群状态相关的，保存在内存里的整体情况。

但是，关于 registry 还有一个需要考虑的地方：严格的和非严格的模式。到目前为止，本书仅仅描述了严格的 registry 模式，因为这是安全的，能够保证正确的逻辑。不幸的是，严格的 registry 有一些令人烦恼的限制：在所有 slave 要么完成注册要么超时之前，用户都无法添加新的 slave（通常，slave 重新注册到新选举出的 master 上有 90 秒的安全时间）。但是，如果使用非严格的 registry，那么框架必须处理核对期间特别复杂的场景。下节详细讨论可能发生的所有情况。

故障转移期间的核对

到这里，大家应该很满意 Mesos 会保证状态一致性的设计思想。好吧，还不应该满足，因为还有一个没有探讨过的问题：在短暂故障期间如何保证框架和 Mesos 集群的同步？比如，如果 master 将一个 slave 标记成 dead，随后 master 崩溃了，从而没能通知调度器所有该 slave 上的任务都进入了 LOST 状态（因为其 slave 失活了），这时会发生什么呢？毕竟，主实例上任务相关的状态 100% 都来自于 slave 的状态，新的主实例完全不知道有过什么状态更新，或者哪些状态更新还没有发送出去。

解决这一难题的方案是核对特性。核对是调度器如何和 master 一起检查调度器所认为的集群状态是否和 master 所认为的集群状态完全匹配。允许调度器在任何时间，给 master 发送 reconcileTasks——这是一个 RPC 调用，允许调度器向 master 提供所有任务及其状态信息的列表，并且 master 会向调度器提供修正信息，告诉调度器它所知道的有更新状态的任务。通常，核对会等到没有 slave 向 master 报告时发生（可能会发生在 master 故障转移期间）。下文还会介绍非严格的 registry 模式是如何导致这里的情况更加复杂的。 128

永远提供 Slave ID

确保向 reconcileTasks RPC 提供 slave ID，即使这是可选参数。如果选择不指定 slave ID，可能会等待两分钟才能得到响应，因为 Mesos 会等到所有注册了的 slave 都报到了或者超时了。另外，可能会错误地接受到某些任务的 TASK_LOST 消息，并且随后发现这些任务又复活了。与其要处理这样长时间的延时以及特殊的场景，不如每次核对都提供 slave ID。

任务核对只能提供非终止状态的任务的细节。当然，核对的主要原因是确定任务是否还在运行，或者已经进入了终止状态。无论何时只要任务进入了终止状态，该任务的核对

就会一直认为其 LOST 了，而实际上它可能进入的是其他终止状态将核对信息和其他信息（比如外部服务的任务成功后可能存储了一些数据）组合起来，来确定已完成任务的真实结果，这是框架开发人员的职责。

核对已完成任务

如果之前看到过某个任务处在不同的终止状态，那么要小心，需要一直忽略来自核对的 TASK_LOST 消息。

只要任务没有进入终止状态，核对都能得到任务的最新状态（如果任务还没有发送任何状态更新，无论最新的状态更新是什么，都会得到 TASK_STAGING）。当某个任务终止后，核对就会无效。但是，当 master 不知道某个任务的状态时，行为会有些诡异。这里需要考虑如下情况：

任务未知，但是其 slave 是已知的

这时，核对机制会报告该任务已经 LOST 了。这是符合逻辑的，因为这意味着任务已经进入了终止状态。

任务未知，slave 在 registry 里，但是还没有重新注册

当 master 的故障转移发生在 slave 有机会报到之前时，会遇到这种情况。Mesos 会简单等一段时间，直到它连接上 slave，再发送核对请求的响应。原因是 master 不知道任务上发生了什么，因为前一个 master 崩溃了。

129 *任务未知，slave 也不在 registry 里*

如果每次都向 reconcileTasks 提供 slave ID，并且使用的是严格的 registry，那么永远都不需要处理这种场景。当这样的情况发生时，Mesos 无法知道是否曾经连接上过 slave，因为该 slave 可能存在，但是还没有包含进 registry 里。因此，Mesos 会报告该任务为 LOST，但是如果这之后 slave 完成了注册，那么用户可能会看到该任务的其他状态更新。任务可能又变成 RUNNING 状态而复活了，或者变成 FINISHED、FAILED 状态而完成了！使用严格的 registry 模式可以完全避免这种情况的发生，因此强烈推荐使用严格模式。

容器机

众所周知，Mesos 全面支持 Docker。但是这意味着什么呢？在命令行里运行 docker run... 就可以使用 Docker 了。还需要做什么？因为 Docker 本身想管理整个容器，从

chroot、命名空间到整个命名空间的 cgroup，它会和默认的 Mesos 容器发生冲突。因此，Mesos 添加了容器机的支持，一种可插拔的机制，让 Mesos 的容器机子系统可扩展：最初 Mesos 的基于 LXC/cgroup 的容器被引入到容器机 API 里，Docker 是添加的第一个新的容器机，现在也有了全面的文档协议，介绍如何添加新的容器机，比如 KVM 虚拟机。

使用 Docker

为了使用 Docker 容器机技术，必须将其包含进 Mesos slave 的命令行里。比如，`mesos-slave --containerizers=docker,mesos...` 允许在该台 slave 上使用 Docker 和 Mesos 容器。

可能还想增加执行器的注册超时时间，这样 Mesos 不会在容器还在下载的时候就认为容器发生了故障。一开始可以设成五分钟，确保有足够的时间下载 Docker 镜像。所以，slave 命令行类似：

```
mesos-slave --containerizers=docker,mesos \

--executor_registration_timeout=5mins ...
```

使用带有应用程序的 Docker 非常简单——一旦启用了对 Docker 的支持，只需要设置 `TaskInfo` 或者 `ExecutorInfor` 里的 `container` 字段（类型为 `ContainerInfo`）。

令人困惑的是，消息 `CommandInfo.ContainerInfo` 并不是正确的消息——需要在带有 Docker 相关字段的 `mesos.proto` 里设置最高级别的 ContainerInfo。

130

要想使用 Docker，需要将 `ContainerInfo` 里的 `type` 设置为 `DOCKER`，并且将 `docker` 字段设置到 `ContainerInfo.Docker` 消息的一个实例里，该消息的 `image` 属性设置为 Docker 镜像的名称（比如 `myusername/webapp`）。这里可以配置很多 Docker 参数，比如是使用 `HOST` 还是 `BRIDGE` 网络，映射使用哪些端口或者额外的 Docker 命令行参数。如果想让 Docker 容器使用 Dockerfile 里指定的 `docker run ...`，还必须将 `TaskInfo` 的 `CommandInfo` 设置成 `shell=false`。如果设置成 `shell=true`，就需要禁用 Dockerfile 里的 `run`，指定的 command 会由 `sh -c "<command>"` 来运行。

当启动 Docker 容器机任务时，slave 会首先获取（并且解包）沙箱里所有指定的 URI，并且将 Docker 镜像拉取到本地。然后，slave 通过运行 `docker` 启动 Docker 镜像。

docker 命令的 HOME 环境变量指向该沙箱，因此可以通过获取到的 URI 来配置 Docker(详见下面的注意事项)。在 Docker 镜像里可以使用该沙箱，其路径保存在 MESOS_SANDBOX 环境变量里。最后，Docker 的 stdout 和 stderr 会被重定向到 Mesos 沙箱里名为 stdout 和 stderr 的文件上。

高级 Docker 配置

必须记住的一点是，Docker 容器机总是会尝试从 registry 里拉取 Docker 镜像。这意味着无法使用仅在本地安装了的 Docker 镜像——必须在某个地方部署该镜像。如果想要使用私有 registry，可以提供一个 .dockercfg 文件。该文件由一个 URI 指定，这样 Mesos slave 就能够使用其自动获取 URL 的功能将 .dockercfg 文件拷贝到 Docker 进程所使用的 HOME 目录下。

相同的 API 也适用于基于 Docker 的执行器，唯一不同之处在于，执行器代码实际上可以在 Docker 容器内运行。要实现这一目的，需要完成上文所述的所有事情，但是是在 ExecutorInfo 消息里，而不是 TaskInfo 消息里。

131

新的 Offer API

选择 Mesos 正是因为它能够提供集群内分配工作负载的灵活度，帮助用户更加高效地使用基础架构。但是有时候，用户会想在 Mesos 上指定更为长期的分配决策。比如，想要预留某些机器来保证某个应用程序的能力，哪怕在其不是运行中任务的时候。当然，有 slave 预留的功能（详见“静态和动态 slave 预留”部分），但是这里新的 API 允许用户编程实现预留，而无须关闭某个 slave。我们需要一种方式来管理集群上这些动态创建的预留，这也正是该 API 所解决的问题。

示例 4-5 里首次使用的 launchTasks API 对于一些框架会有一定的负面影响：一旦任务退出，无论是正常退出还是因为故障退出，其他框架就可以声明使用该任务的资源。本节会详细介绍这个新的 offer API，它允许框架超越其任务的生命周期，而动态预留和释放资源，并且该 API 还带来了一些额外的特性。

框架动态预留 API

动态预留是一个新特性，如果框架需要在重启期间保留其资源（比如数据库以及带有严格 SLA 的系统），可以随时启用 slave 预留。不用使用 launchTasks API，而是使用 acceptOffers API。示例 6-1 展示了 launchTasks 和 acceptOffers 调用。

示例6-1 launchTasks和acceptOffers比较

```
driver.launchTasks(
    Collections.singletonList(offer.getId()),
    Collections.singletonList(taskInfo)
);

// 对比

Launch launch = Offer.Operation.Launch.newBuilder() ❶
                .addTaskInfos(taskInfo)
                .build();
Operation launchOp = Offer.Operation.newBuilder() ❷
                     .setType(Offer.Operation.Type.LAUNCH)
                     .setLaunch(launch)
                     .build();

driver.acceptOffers(
    Collections.singletonList(offer.getId()),
    Collections.singletonList(launchOp), ❸
    Filters.newBuilder().build() ❹
);
```


❶ 首先，构造想要使用 offer 处理的实际命令——启动这些任务。

❷ 随后，构造发送给 Mesos 的操作——由 `Offer.Operation` union 表述的所有不同命令。注意 `Operation` 是一个 protobuf 标记 union（详见“理解 mesos.proto”部分）。

❸ 这里，仅需要在 offer 上做一次操作。可以选择指定一些操作列表，还可以组合预留资源，创建持久化卷，以及在单个消息里启动任务。

❹ `acceptOffers` 要求用户传入 `Filters` protobuf。可以使用默认值，默认不做任何筛选。

除了使用 `acceptOffers` 启动任务，框架还可以用动态预留来预留该任务的资源。当该任务终止时，其使用的资源会被仅对该框架角色有效的预留独占。这意味着用户可以启动生产环境工作负载，无论其是否有预配置的静态预留，机器重启以及临时故障都不会永久破坏其能力（对于必须维持严格 SLA 的服务）或者功能（对于需要保留数据的服务，比如数据库）。下面的示例里会添加预留这些资源的功能。向在这些 offer 上执行的操作列表里添加另外的 `Offer.Operation`，如示例 6-2 所示。

示例6-2 在单个acceptOffers里预留资源并启动任务

```
Offer.Operation.Launch launch = Offer.Operation.Launch.newBuilder() ❶
                                .addTaskInfos(taskInfo)
                                .build();
Offer.Operation launchOp = Offer.Operation.newBuilder()
                           .setType(Offer.Operation.Type.LAUNCH)
                           .setLaunch(launch)
                           .build();

Offer.Operation.Reserve reserve = Offer.Operation.Reserve.newBuilder() ❷
                                  .addAllResources(taskInfo.getResourcesList())
                                  .build();
Offer.Operation reserveOp = Offer.Operation.newBuilder()
                            .setType(Offer.Operation.Type.RESERVE)
                            .setReserve(reserve)
                            .build();

133 driver.acceptOffers(
        Collections.singletonList(offer.getId()),
        Arrays.asList(reserveOp, launchOp), ❸
        Filters.newBuilder().build()
    );
```

❶ 和之前一样，构造启动操作。

❷ 还需要构造预留操作。这里，从启动了的任务里简单拷贝所有资源，这样任务的预留能够跨任务的生命周期持久化存在。

❸ 将多个操作传递进 acceptOffers。这和等待 master 通知并且重新提供预留是等价的，但是减少了跨网络的来回通信的次数。

上文已经介绍了如何预留资源和启动任务，还剩下一个逻辑性问题：如何判所提供的资源是否来自于已有的动态预留？资源带有额外的字段 reservation，可以据此判断这些资源是否已经被预留，被谁预留了，如示例 6-3 所示。

示例6-3 查询资源的预留状态

```
if (resource.hasReservation()) { ❶
    resource.getReservation().getPrincipal(); ❷
}
```

❶ 如果资源是预留的，该调用会返回 true。

❷ 如果资源是预留的，就可以得到预留它的管理员名称。管理员通常就是框架的 user。

在 offer 里处理预留

当使用动态预留时，可能需要修订计算 offer 里有多少资源的代码。回过头看看示例 4-10，注意示例里实际每次计算 CPU 或者内存资源时使用的是 +=。这样做能够确保可以使用预留和非预留资源来启动任务。

一旦 slave 死亡了，预留会自动解除。如果框架跟踪所有大额预留，就可以使用 `slaveLost` 回调函数来判断旧的 slave 上的预留什么时候开始不复存在了。如果不再需要预留，可以使用 `acceptOffers` 里的解除预留操作，如示例 6-4 所示。 <134>

示例6-4 解除资源的预留

```
List<Resource> resources = new ArrayList<>();
for (Resource r : offer.getResourcesList()) { ❶
    if (r.hasReservation()) {
        resources.add(r);
    }
}
Offer.Operation.Unreserve unreserve = Offer.Operation.Unreserve.newBuilder()
                                      .addAllResources(resources) ❷
                                      .build();
Offer.Operation unreserveOp = Offer.Operation.newBuilder()
                              .setType(Offer.Operation.Type.UNRESERVE)
                              .setUnreserve(unreserve)
                              .build();

driver.acceptOffers(
    Collections.singletonList(offer.getId()),
    Collections.singletonList(unreserveOp),
    Filters.newBuilder().build()
);
```

❶ 找到所有之前预留的资源。

❷ 在这里，解除之前预留的所有资源。

预留是永久的

如果在任务完成时不解除资源的预留，Mesos 会永远将这些资源留在一边，直到用户解除预留为止。一旦 HTTP API 开发完成后，动态预留功能就会被去掉测试版本的标签。HTTP API 让 Mesos 集群的运维人员可以代替框架解除资源的预留。

动态预留 API 是很强大的工具，可以帮助在 Mesos 上构建更加健壮的应用程序。master 上还有 HTTP API 作为管理动态预留的另一种方式（详见“动态预留”部分）。该 API 使得随时跨 Mesos 集群变更预留变得很容易。它还可以帮助解除框架在其最后一次关闭时

忘记了的资源的预约。动态资源仅仅是 acceptOffers API 的开始，下面介绍持久化卷 API。

135

数据库使用的持久化卷

直到最近，Mesos 都仅仅能够运行无须向磁盘存储数据的服务。这是因为没有方法预留所需的磁盘块。从 Mesos 0.23 版本开始，可以预留磁盘了。

第 1 章中曾经说过，可以将 Mesos 当作一个部署系统。如果 MySQL 数据库能够自动将自身备份，并且按需创建新的副本，是不是很好呢？或者如果拥有一个简单的，自服务的 REST API，能够创建新的 Riak 和 Cassandra 集群，又会怎么样呢？为 Mesos 构建数据库框架的工作从 2014 年就开始了。这些框架的问题是每个主机都必须创建特别的数据分区，并且在 Mesos 之外加以管理。使用持久化卷，类似 Apache Cotton（MySQL 所用）以及 Cassandra 和 Riak Mesos 框架的项目就都能够独立启动和维护了。

在 Mesos 的设计里，磁盘空间是短暂的，并且是按任务隔离的。这通常是一件好事，除非用户想要持久地保存数据。要解决这个问题，Mesos 引入了一个新的磁盘资源的子类型，称为 volume。volume 是分配给一个任务的磁盘块，并且挂载在特定位置。完成这一功能的 API 和挂载主机卷的 Marathon API（详见“挂载主机卷”部分），几乎完全一致。用户甚至可以创建不持久的卷，这在想将多个独立磁盘暴露给 Mesos 时会很有用。下面研究一下如何创建并且使用持久化卷。

有两个 acceptOffers Operation 用来创建以及销毁持久化卷。不出意外地，它们称为 Create 和 Destroy。仅仅能够在已经被预留的磁盘资源上创建持久化卷。通常，用户会预留资源，创建卷，并且在单个 acceptOffers 里启动任务，如示例 6-2 所示。

持久化卷资源和常规磁盘资源（详见“资源”部分）一样，但是它带有字段 disk，设置为合适的 DiskInfo。DiskInfo 给该持久化卷命名，这样它能够挂载上，名字为嵌套的字符串子字段 persistence.id 的名称。DiskInfo 的 Volume 必须使用 RW 模式（因为 Mesos 0.24 只支持 RW）。Volume 的 container_path 字段会指定容器在任务沙箱里的挂载位置。

持久化卷 API 是很新的功能，因此还没有任何生产环境框架用到它。它也有一些限制，比如卷必须一直挂载为 RW，并且没有办法暴露多个磁盘，也没有任何磁盘或 I/O 隔离。即使添加了新特性和功能之后，也会保证该 API 的后向兼容性。因此，类似 Apache Cotton 的项目已经在其代码基里集成了持久化卷。

本章小结

Mesos 为很多不同的用户场景都提供了精妙的，考虑周全的 API。本章首先从高层探讨了 Mesos 的内部架构，帮助大家理解为什么要像现在这样构造 Mesos。随后深入分析了 Mesos 是如何维持内部和外部的一致性的。通过理解一致性模型，我们可以构建出更加健壮并且容错的框架。

接下来，本章介绍了用来同步框架和 Mesos master 的核对流程。理解了核对流程的机制，就能够理解如何最终保证 Mesos 集群内的一致性，会发生哪些类型的不一致性，以及如何修正这些不一致的情况。

然后介绍了 Mesos 和 Docker 的集成，Docker 是一种流行的应用程序的容器化技术和部署格式。将容器化技术集成到框架里，用户就可以利用更为宽广的生态系统，比如所有 Docker 化的应用程序。

最后一起研究了动态预留和持久化卷，它们是由新的 `acceptOffers` API 引入的特性。动态预留帮助用户更容易地保证关键工作负载的能力。持久化卷让用户可以为 Mesos 构建数据库框架，Mesos 可以在任何不可预见的故障和错误发生并且影响整个系统时，使数据持久化。

至此，本书已经介绍了很多主题和技术，有助于现在在 Mesos 上构建应用程序。下一章会介绍以后 Mesos 还有哪些值得期待的发展。

第 7 章

137

Mesos的未来

Mesos 已经有了长足的发展。它从加利福利亚大学伯克利分校的研究生项目起步，至今已经运行在跨越成千上百台机器的生产环境之上，吸引了不计其数的开发人员投身到其生态圈中。本书已经讲解了如何在 Mesos 上构建应用程序，但是 Mesos 会走向何方呢？本章会介绍 Mesos 生态系统里刚刚发起的几个项目，这些项目已经开始彰显出其重要性，并且可能会成为 Mesos 很有价值的特性。

多租户工作负载

在讨论多租户之前，先看看这个问题：烦人的邻居。烦人的邻居是现实生活中的问题，也是多用户（租户）分布式系统里的实际问题。当一套公寓建筑物的墙太薄时，你就能够透过墙壁听到邻居的巨大音乐声。类似地，当系统无法提供充分的隔离（就像厚墙壁）时，应用程序的性能就会受在同一台机器上运行的其他应用程序的影响。比如，多个 CPU 敏感的应用程序运行在同一台机器上，可能会竞争使用每个 CPU，导致整体性能下降。从而让用户或者集群运维人员很难预测其应用程序的性能——反之，如果这些应用程序运行在容器里，则都能够保证它们得到 CPU 的一部分，从而降低性能上的不可预见性。

多租户技术指的是当必须在很多用户间共享单一资源（本书场景里是 Mesos 集群和其 slave 上的资源）时，一些用户意外地独占了本该共享的资源时所出现的相关问题。用户希望系统在设计上就是支持多租户的，因为这样 Mesos 能够更容易地管理资源的隔离，而不用依赖于用户去仔细编写互相合作的应用程序。

138

如今，很多 Mesos 的安装受益于容器化技术，将不同的应用程序隔离开。以后，会有越来越多的 Mesos 集群将不同的用户或者客户隔离开。很多大型企业深受多租户问题的困扰。比如，很多公司为每个需要集群的团队管理并且运行单独的 Hadoop 集群。为什么

他们不共享一个大集群呢？通常来说，如果每个团队无法预测某个时刻他们能够使用多少集群资源，那么他们就无法得到所需的服务级别和性能。单独集群将团队互相隔离开，确保每个团队总是能够完全使用其所需的资源。系统性隔离是解决烦人邻居问题的一种方案，但是，因为没有共享，公司必然需要购买更多的计算机。

Mesos 允许很多用户共享相同的物理硬件（或者 VM），来服务多租户的工作负载，使用 Mesos、Linux 容器提供了隔离性。Linux 容器是一种基于 cgroups 的技术，由 Google 开源，可以帮助实现同一台机器上多种工作负载的高性能隔离。通过 CoreOS、Docker 和 Mesos 这样的项目的共同努力，Linux 容器已经从一种有趣但是很难使用的技术成长为强大、流行、轻量级的隔离机制。虽然 cgroups 最初仅仅支持 CPU 和内存的隔离，但是正有越来越多的 Linux 内核子系统在集成进来。比如，2014 年 7 月，Mesos 实现了和 Linux 网络隔离堆栈的深度集成。这能够配置 Mesos 在集群上运行的容器间控制并且隔离网络带宽的使用。目前，有项目正致力于隔离容器间的磁盘 I/O 的使用。随着时间的推进，Mesos 会添加越来越多的隔离特性，这会进一步降低多租户工作负载的烦人邻居问题的困扰。

还有一些 Mesos 框架正在构建中，意图从更高层解决多租户问题，帮助企业更容易更好地隔离共享 Mesos 集群上的不同用户。比如，Myriad 由 eBay 在 2014 年启动，在 Mesos 上运行 YARN。YARN 是 Hadoop 2.0 的资源管理器。使用 Myriad 在 Mesos 上创建新的且完全隔离的 YARN 集群仅仅需要一个 REST API。因此，如果部署了 Myriad，当某个新团队需要 Hadoop 集群用来实验或者作为生产环境时，几乎没有任何管理性或者运维性的开销。另一个多租户 Mesos 框架是 Cook，由本书作者编写，Two Sigma 在 2015 年将其开源。Cook 是一种抢占式作业调度器，它允许任何用户使用可用的尽可能多的资源，但是当之后其他用户所要求的能力增长时，会自动缩小该用户的资源。

Mesos 支持多租户的另一种方式是通过资源预留。可以使用预留来启用以及强制多租户环境里的最小份额，让其成为多租户环境里的最强势的资源保障，但是这么做是以牺牲动态灵活性为代价的。如果通常来说用户不会占用整个 Mesos 集群，那么可以不用配置预留，而是让所有框架随意自由地伸缩。当一些用户必须满足一定的生产环境关键的服务级别协议时，可以使用预留保证他们能够占有特定的资源。从 0.25 版本开始，Mesos 添加了 RESTful HTTP API，这样，集群运维人员能够随时为用户创建预留，从而简化了保证特定用户和工作负载所占资源的方式。还有一个项目是要向 Mesos 添加 RESTful 配额 API，这样运维人员能够让 Mesos 基于每个角色限制使用多少资源这样的高层级规则为其找到预留。

Mesos 提供了很多特性，帮助构建更加健壮的多租户系统。它和 Linux 内核的资源隔离机制集成，提供集群上任务间的底层隔离。框架作者也要了解这个问题，越来越多的框

架正在试图简化多租户集群的管理。资源预留是解决多租户问题的最为强大的工具：它绝对保证某个租户的资源，以阻止其他用户访问这些资源为代价。现在，如果有一种方式，使资源可以被预留，但是在当前不需要时仍然可以共享，是不是很“酷”呢？下节会介绍 Mesos 在 0.23 版本里添加的超配特性。

超配

集群利用率 100% 意味着什么？从某种意义上来说，Mesos 集群上没有任何空闲资源时，就可以说达到了 100% 的利用率。但是，即使某个框架为某个任务（比如一个 web 服务器）预留了资源，该任务可能并不会完全使用所有这些资源。实际上，在大多数集群上，实际使用率仅仅有 10% ~ 30%。为了解决这个问题，Mesos 推出了超配特性。该特性允许 Mesos 集群自动使用预留但是未使用的资源。[1]

要理解超配，首先需要定义 slack。slack 是指认为被使用了的资源和实际使用的资源之间的差（图 7-1）。降低 slack 是所有人的共同目标：slack 纯粹是浪费，这里的资源本来可以做一些有产出的事情，却白白空闲着。Mesos 集群里有两种类型的 slack：

分配 slack

> 分配 slack（allocation slack）是集群上的可用资源和框架预留资源之间的差。Mesos 从设计上就可以高效解决这一类型的 slack，通过向所有连接着的框架重复重新提供资源来实现。这样，如果某个框架不需要或者无法利用某些资源，其他框架就有机会使用这些资源。一些框架，比如 Spark，通过启动很多使用少量资源的小型任务来利用这一点，这样它们能够在很多机器上得到很小的资源分配，从而提高集群的利用率，并且将这些资源贡献给用户。

140

使用 slack

> 使用 slack（usage slack）是预留资源和实际使用资源之间的差。比如，如果 web 服务器预留了两个 CPU，但是在没有太多待处理请求的非高峰时段，它几乎不用使用任何资源。超配特性帮助 Mesos 降低这一类型的 slack。

从某些方面来看，Mesos 成了自身成功的受害者：Mesos 集群里能够大幅降低分配 slack，商业生产环境集群的分配 slack 为 5%~15%，这比前面所述系统的 20%~30% 可低得多。因此，企业联盟，包括 Twitter、Mesosphere 和 Intel，开始构建一种能够降低 Mesos 使用 slack 的系统。它们工作的产出就是 Mesos 超配系统，在很多资源的使用 slack 上应用控制理念。

1 Mesos 超配特性是基于 Google 的 Heracles 系统，是解决低实际使用率问题的方案。

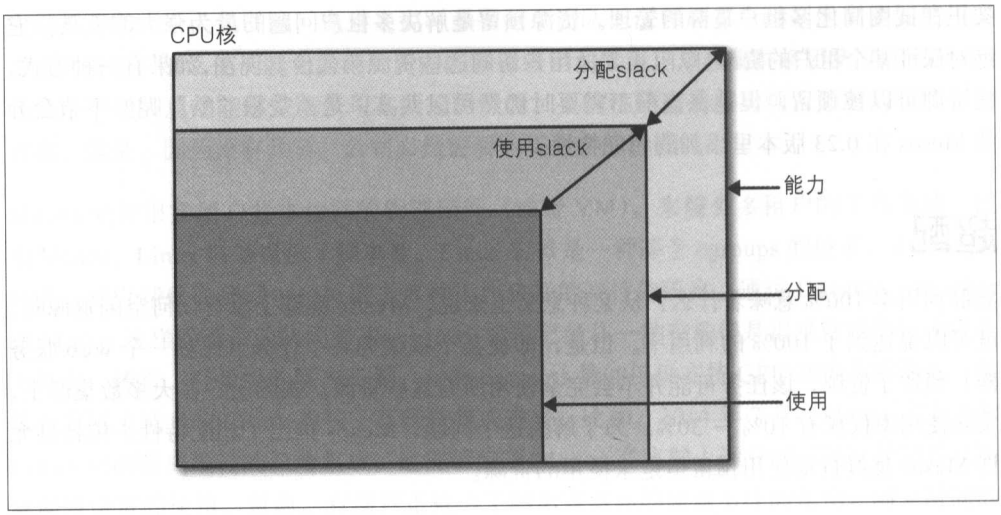


图7-1 CPU、内存的分配slack和使用slack

要启用超配，Mesos为此添加了一种新类型的资源offer：可撤销offer。可撤销offer和常规offer几乎完全一样，除了这一点：在可撤销offer上启动的任务可以随时被Mesos“杀死”。默认情况下，框架不会接受任何可撤销offer。在注册时向FrameworkInfo里添加REVOCABLE_RESOURCES功能就能选择接受可撤销offer。如果
141 Mesos集群还被配置为启用超配机制，那么任何选择接受可撤销offer的框架在看到常规offer时，也会看到另一种类型的offer，这些特别的offer带有revocable字段，这意味着它们在任何时间都可以被取消。注意当前执行器必须有可撤销的或者常规的资源，无法在相同的执行器上启动可撤销和非可撤销任务的混合体。

超配系统包含两个可插拔组件：资源估算器和服务质量（QoS）控制器。资源估算器的工作是向其slave报告运行着的任务里有多少可用slack，从而slave能将这些额外资源作为slack发布出去。QoS控制器的工作是跟踪使用slack。当使用slack降低，低于分配给可撤销资源的资源时，QoS控制器就会开始“杀死”可撤销任务，确保集群能够保证预留资源的提供。

在Mesos 0.23里，只有一个固定的资源估算器，集群管理员可以让所有slave都发布其固定的额外资源。当大多数任务请求的是一个或者两个CPU，但是却仅仅使用了其资源的5%～10%时，该功能就很有用了。固定的资源估算器会允许每个slave上调度比实际CPU更多的任务。Mesos 0.23仅仅发布了一个不工作的QoS控制器。

Mesosphere和Intel还构建了Serenity，这是一个精妙的控制系统，周期性地衡量每个slave上的使用slack，以便集群能够利用这些资源。Serenity还知道如何估算烦人邻居问

题的影响，如何区分正在启动的任务和达到稳定状态的任务，以及其他一些优化集群使用率所必需的实用的调整方案。

超配是一个强大的新特性，可以帮助大型集群多利用 10% ~ 40% 的资源。超配引入的可撤销资源并不仅仅在分配 slack 资源时有用，它们还构成了优先任务的基础。

数据库和 Turnkey 基础架构

现今，为永远可用的分布式系统创建一套完整的基础架构是很有挑战性的一件事。你需要搭建数据库、web 服务器、大数据文件系统、分析工具、监控系统以及报告生成流程。这需要大量的工作——困难不仅仅来自要在各个领域不计其数的技术和解决方案中完成选型，还来自每种工具的安装和配置都有属于自己的特别方式。创建能够重复安装和配置的流程，通常是最大的痛点。另一方面，常常必须编写大量定制的部署代码，才能确保能够可靠地重建整个基础架构。另外，这么做也相当昂贵且费时，因此这样的工作一般都会被推迟，直到系统有时间丰满成熟为止。Mesos 正位于改变部署分布式应用程序基础架构方式的交叉路口。

Mesos 一直擅长运行面向计算的任务，比如 web 服务器和分析式框架。在版本 0.23 里，Mesos 最终添加了对管理集群磁盘的支持。因此，很多面向数据的应用程序现在也能够成为 Mesos 的框架了。比如，已经能够看到在 Mesos 集群上使用 Cassandra[1]、HDFS[2]、和 Kafka[3]。2016 年早期，越来越多框架使用 Mesos 持久化磁盘 API，Mesos 的生态系统终于为基于磁盘的应用程序提供了大量健壮的可选方案。令人激动的项目之一是 Apache Cotton（之前的 Mysos），它为 MySQL 集群提供了强大的自动化方案。

2016 年，大家最终能够简单地安装一个 Mesos 集群，然后让 Mesos 集群启动数据库、监控系统、web 服务器和分析引擎。这极大地减少了搭建并且运行可扩展、高可用、自修复集群所需的工作量。因为 Mesos 框架是运行着的，所以能够适应变化着的集群条件的程序，它能够自动处理机械化的数据库维护和其他标准的集群运维任务。通过使这些费时的任务自动化，运维人员将能够更多地关注其他问题，比如应用程序调优和增加利用率。

基于容器的 IP

Mesos 的另一个令人激动的特性是基于容器的 IP。这个功能想要解决如下两种问题：

1 Cassandra 是高性能的 NoSQL 数据库，其极度的高可用性广受欢迎。
2 HDFS 是 Hadoop 文件系统，也是最流行的大数据文件系统之一。
3 Kafka 是高性能的复制队列，帮助大数据发布 / 订阅工作负载的扩展。

1. 想使用纯 DNS 名称，让浏览器自动找到 Mesos 集群上的任务（比如 mytask.mesos.mycompany.com）。

2. 想要更容易地将设计运行在整台机器上的应用程序引入到 Mesos 上（比如 Cassandra、Riak，以及其他数据库）。

Mesos 已经提供了 DNS 的解决方案，称为 Mesos-DNS。Mesos-DNS 从 master 上读取所有任务的元数据，对于那些指定了服务发现信息的服务，或者仅由字母数字组成名称的
143
服务，Mesos-DNS 会生成 DNS 记录，这样就可以通过标准机制发现这些任务。但是，这里有一个难题：当任务需要侦听一些端口时，大多数 Mesos 框架会基于 slave 上的可用端口来动态分配端口。因此，即使知道该主机的 DNS 名称，浏览器也仅仅会尝试端口 80（HTTP）和 443（HTTPS）。因此，浏览器无法真正连接到运行着的应用程序上。现在，Mesos-DNS 解决了这一难题，它还提供 SRV 记录，包含了服务运行的端口号。不幸的是，浏览器不支持 SRV 记录，所以需要使用 Nginx 做中转。[1]

基于容器的 IP 系统解决了这一问题，它给每个执行器分配属于自己的特定 IP 地址。这样，任务的 DNS 入口指向的是执行器的 IP 地址，并且随后任务可以绑定到这个私有 IP 地址想要使用的任何端口上。每个执行器容器都有属于自己的 Linux 网络命名空间。为每个容器创建单独的网络堆栈，这样主机能够将流量路由到合适的容器里。另外，每个网络堆栈都能够具备适用于它的服务质量，从而每个容器都能够分配到有保障的网络带宽。该特性允许 Mesos 将 IP 地址作为最高级别的资源来管理，就像 CPU、内存和磁盘一样。

本章小结

Mesos 生态系统里有很多令人激动的新特性。Mesos 是从无到有构建起来的，提供了构建多租户系统所必需的强大隔离保障，并且我们已经开始看到像 Myriad 和 Cook，以及网络隔离器这样的特性的涌现。

随着多租户应用程序的增长，Mesos 正在开发精妙的超配特性，帮助达到 Google 所能达到的集群高使用率。该超配特性很容易使用（如果想要固定的超配），并且也正在稳步推进高级动态适应的超配的开发。

持久化磁盘和基于容器的 IP 是最后介绍的特性，它们让 Mesos 能够提供完整的基础架构管理系统，这也是其他产品，比如 Openstack 所致力的方面。Mesos 集群能够管理企业的所有基础架构，包括分析、数据库和服务。基于容器的 IP 地址使得将要求特定端口范围的遗留应用程序迁移到 Mesos 上变得更容易，同时无须构建复杂路由和代理层，也

1 可以配置 Nginx 使用 SRV Router，基于 SRV 记录路由入站域名。

可以很容易地将 Mesos 应用程序暴露给终端用户。

Mesos 正在占领数据中心，它的开源生态系统给所有人带来了轻量级容器化技术，编排，以及简单的、中央化的管理系统。

索引†

†： 中文版书中切口处“▭”表示原书页码，便于读者与原英文版图书对照阅读，本书的索引所列页码为原英文版页码。

E

F

G

H

N

O

P

Q

R

关于作者

David Greenberg 是 Two Sigma 的首席架构师，他负责公司交易策略所用的分布式计算环境。David 有强烈的学习欲望，自学了俄语和汉语，并且他很喜欢练习厨艺。他也是一个调度独占作业的开源 Mesos 框架——Cook 的设计师。

版权页

《用 Mesos 框架构建分布式应用》一书的封面动物是印度巨松鼠（Ratufa indica），也称为巴拉马尔巨松鼠。印度巨松鼠栖息在树上，几乎从不离开它们树顶上的家，生活在印度次大陆的森林里。它们对森林生态系统的种子传播起着重要的作用。

印度巨松鼠的身体能长到 36 厘米高，它们的尾巴则能长到身体的几乎两倍长。它们的长尾巴提供了很好的平衡感，帮助印度巨松鼠在树枝间快速移动。它们的皮毛颜色与众不同，背上是黑色的，与奶油色或米黄色的头部、尾巴和腹部形成了鲜明的对比。

印度巨松鼠通常独居或者两两居住。它们的食谱包括水果、鲜花、坚果、树皮和昆虫。它们的天敌是巨猫、掠夺性鸟类和蛇。虽然目前印度巨松鼠还没有濒临灭绝，但是由于森林的采伐，它们的数量也正在减少。

O'Reilly 封面的很多动物都是濒危动物，它们对于世界而言很重要。可访问 animals.oreilly.com 了解你可以提供哪些帮助。

O'REILLY®

用Mesos框架构建分布式应用

Apache Mesos如何让你的企业与众不同？通过本书，你将了解集群管理器如何管理数据中心资源，并且提供实时的API，来与整个集群完美交互。你将学习如何将Mesos作为一种和Ansible或者Chef类似的部署系统，如何将其作为一种执行平台来构建及托管Hadoop这样的高层级应用程序。

作者David Greenberg向大家阐述了Mesos如何将数据中心当作单个逻辑实体来管理，完全无须给任何应用程序指派固定的机器集。你将能够快速理解Mesos为什么是终极的DevOps工具。

- 理解Mesos架构，并且学习如何在集群内管理CPU、内存及其他资源
- 在Mesos上使用Marathon构建应用程序，Marathon是Mesos上托管服务的平台
- 为Mesos创建全新的，符合生产环境要求的框架
- 编写自定义执行器，提供Mesos调度器和worker之间的丰富交互
- 深入高级话题，包括核对流程、Docker集成、动态预留，以及持久化卷
- 学习当前的一些Mesos项目，它们很可能会成为Mesos将来的特性

“《用Mesos框架构建分布式应用》是介绍构建前沿Mesos框架中非常好的一本书。David已经大规模使用了Mesos框架，凝聚了这些宝贵知识和经验的这本书是业界之幸。”

——**Benjamin Hindman**
Apache Mesos的创造者及Mesosphere的联合创始人

David Greenberg是Two Sigma公司的首席架构师，他负责公司交易战略的分布式计算环境。他也是Cook的设计师，Cook是一个开源的Mesos框架，用来做抢占式作业的调度。

PROGRAMMING TOOLS

图书分类：大数据

责任编辑：徐津平

Broadview®
www.broadview.com.cn

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

ISBN 978-7-121-30677-8

定价：55.00元